Optimization methods in remote sensing and geoinformatics

Ph.D. Dissertation

Balázs Dezső

Supervisor: Dr. István Fekete

Eötvös Loránd University
Faculty of Informatics

Doctoral School of Informatics
Prof. András Benczúr, D.Sc.

Doctoral program of Information Systems
Prof. András Benczúr, D.Sc.

Budapest, 2012

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to my supervisor, István Fekete for his encouragement, support and collaboration in this PhD thesis. His interest in applied computer science always gave me enthusiasm to work on real-world problems and to develop new algorithmic solutions.

Among my colleagues, I would like to thank István Elek for the deep conversations on the IRIS project and on artificial intelligence. He has shown me that in a successful research project there are people who think freely and others who concentrate on the details. I also say thank to Alpár Jüttner for the common work on the LEMON project. I learned from him a lot about graph theory and software development. When someone works on a public software library, the desire for the clean design and appropriate naming convention becomes part of the personality.

I would also thank István László: his deep practical knowledge in remote sensing was essential during the common research. I am also grateful to Péter Kovács, for his constant engagement to improve either a program code or a common paper. I was also glad that I worked together with Zsigmond Máriás and Roberto Giachetta.

Last, but not least, I am most grateful to my family. My parents, Imréné and Imre Dezső always gave me the biggest support for learning and continuous desire for more knowledge. I also thank to my sisters, Dr. Zsuzsanna Soósné Dezső and Andrea Witzlné Dezső, because they gave me an ambiance where learning is a fundamental necessity. Finally, I would like to thank to my partner, Lilla Kis, who has encouraged me to write my thesis with a lot of patience.

The research project "Remote Sensing Image Analysis" presented in this dissertation is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

1 Introduction

This thesis presents three real world problems and their algorithmic solutions. The first problem is to classify pixels into specific categories in satellite images. The second one is to convert scanned raster maps to vectorized data. The third one is to plan the schedule of agents who visit customers in a city or in a larger region.

Beside these three problems, there is also a fourth topic, which is about the LEMON network optimization library. This library can be used as a basic starting point to implement optimization algorithms, therefore it was widely used for developing solutions for the previous three problems (see Fig. 1.1).

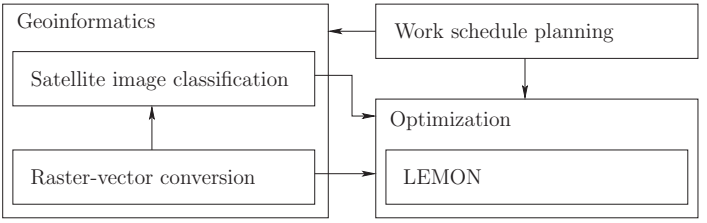


Figure 1.1: Relations between the topics[1]

1.1 Optimization

The optimization, or more specifically mathematical optimization, is the selection of a best element from a set with respect to an objective function. Formally,

- There is given a set A, i.e. the domain of the optimization.
- There is an objective function $f : A \to \mathbb{R}$.
- Find $x_0 \in A$, that satisfies $f(x_0) \geq f(x)$ for all $x \in A$

Due to its general definition, it has countless applications in scientific disciplines, like in engineering, economics or chemistry. In this thesis, we discuss applications related to remote sensing and geoinformatics.

[1]In the figure, the arrow means "using the methods and results of" relation.

Mathematical optimization has several major branches. For example, one of the most studied fields is the linear and integer programming. For the work schedule planning problem, an integer programming model is constructed and solved with column generation method.

The graph theory and network optimization have also important role in the optimization theory. In this dissertation, several graph-based segmentation algorithms are studied and compared by the performance in the satellite image classification problem. Furthermore, the various object types of vectorized maps are also stored as graphs, which helps to maintain the consistency of the map topology.

Geometric optimization algorithms are often used in the image processing methods. For example, Delaunay-triangulation and minimum enclosing rectangle computations were also used in the raster-vector conversion process.

LEMON optimization library. The usage of computers allows solving large-scale optimization problems. In order to make the solution of these problems simpler, several libraries and various tools are implemented, which can used as building-blocks of the optimizer applications.

In Section 5, the LEMON library is presented, which provides a general toolkit for solving optimization problems. The goal of this software library is to provide highly efficient, easy-to-use and well-cooperating software components, which help solving complex real-life optimization problems. These components include graph implementations and related data structures, fundamental graph algorithms (such as graph search, shortest path, spanning tree, matching, and network flow algorithms) and various auxiliary tools (for example, flexible input-output support for graphs and associated data). Furthermore, the library provides a common high-level interface for several linear programming and mixed integer programming solvers.

1.2 Geoinformatics

The goal of geoinformatics is to use information technology to acquire, organize and utilize spatial data. It has countless applications in several fields, e.g. in environmental sciences, economics, agriculture or in public administration.

Classification of satellite images. The remote sensing is a subfield of the geoinformatics, which allows collecting large amount of data from the land surface in a cost-efficient and objective way.

In section 2, the thematic classification is studied, which is a fundamental step in the processing satellite images. The section describes different methods and compares them by quality and performance in crop mapping.

Raster-vector conversion of topographic maps. In Section 3, the raster-vector conversion is described, which is aimed to transform the scanned digital raster maps to vector data. The topographic maps in vector format are usually preferred to the raster maps in GIS applications. In the IRIS project, a generic framework was developed for the raster-vector conversion and several methods were implemented to process specific object types of the map.

1.3 Optimization of work schedule of agents

In Section 4, a problem of optimizing the working schedule and the traveling route of agents is discussed. The optimized schedules can decrease the operational cost of the company and increase the employee satisfaction.

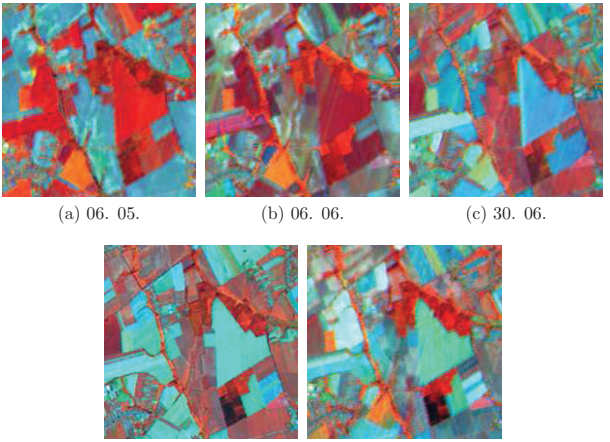
The problem strongly depends also on geoinformatics, because the agents have to travel during the working day, and the optimization of the routes needs spatial data, i.e. road networks and public transportation schedules.

2 Classification of satellite images

The thematic classification is one of the most important tasks of remote sensing, since it is often the initial step of the higher level object recognition.

For the thematic mapping, we use the following inputs:

- one or more satellite images of the same region and in the same spatial system (see figure 2.1)
- the thematic categories, i.e. the classes that the pixels have to be assigned to
- a training reference map, which contains the real categories for a subset of image pixels (see figure 2.2a)



(a) 06. 05. (b) 06. 06. (c) 30. 06.

(d) 27. 07. (e) 17. 08.

Figure 2.1: Subsections of images

The result of classification process is the thematic map, which assigns each pixel to one of the thematic categories. The quality of the classification is determined with a testing reference map, which is structurally identical to

the training data, but contains a different subset of image pixels[2] (see figure 2.2b).

(a) Training data (b) Testing data

Figure 2.2: Training and testing reference data

The classification works on preprocessed images, i.e. they are corrected geometrically and radiometrically. The actual steps of preprocessing can be found in Richards [56].

The thematic classification can be treated directly as minimization of the classification error. For example, the maximum likelihood method minimizes the expected value of the error in its model. In other algorithms, the optimization problems are used in sub-steps, for example to build clusters in the spectral space, or form segments in the image.

On classification problems in remote sensing, the author has mainly been working together with István Fekete and István László. The Institute of Geodesy, Cartography and Remote Sensing (FÖMI) has supported this work both with raising problems and with providing resources. Occasionally, Tamás Pröhle has given support in the statistical problems. Roberto Giachetta participated in the Hungarian National Scientific Student Conference with the co-supervision of István Fekete and the author, where his study "Graph-based methods in thematic classification in remote sensing" achieved 2$^{\text{nd}}$ prize award [31].

During the research project, the author has developed a framework for solving and evaluating classification problems. He has participated in the selection and development of the investigated methods, especially in the analysis of the performance of the graph-based image segmentation algorithms in remote sensing. Several segmentation, clustering and classification algorithms have been implemented and evaluated statistically in the former

[2]In practice, the two reference maps might share some areas when there are not enough reference data for particular thematic categories.

framework. He has also participated in teaching of the course "Analysis of remote sensing images" in the MSc module "Geoinformatics". His research and educational activity has conduced to the current results of the remote sensing research community at ELTE and FÖMI.

2.1 Properties of the satellite images

In this research project, LANDSAT TM and IRS LISS satellite images were used. The LANDSAT 7 TM[3] images [42] have 6 multispectral bands with 30m×30m spatial resolution, and a thermal infrared with 60m pixel size. The satellite has also a panchromatic sensor with 15m pixel size. The LANDSAT 5 TM sensors are almost identical, but there is no panchromatic sensor and the thermal infrared sensor has 120m spatial resolution. The radiometric resolution of the bands is 8bit, i.e. each value is from the range 0-255. The repeat intervals of the satellites are 16 days, so in every year, multiple images are available for the evaluation.

The IRS P6 LISS-III[4] sensor [54] have 4 multispectral bands including also a thermal infrared with 23.5m × 23.5m resolution. It can also operate in panchromatic mode, which provides 5.8m resolution imagery. The radiometric resolution of the bands is 7bit.

The spectral properties, the spatial resolution and the sufficiently frequent returning of these satellites make both the LANDSAT and the IRS images effectively usable in agricultural applications. For the crop mapping, we use composites of near-infrared, mid-infrared and visible red bands. For a particular region, we need a set (4-6) of images created on different days of the year.

2.2 The pixel-wise classification methods

In the pixel-wise classification method, the satellite image is treated as a set of spectral reflectance vectors, i.e. the class of each pixel is decided independently from the other pixels.

2.2.1 Maximum likelihood and Bayes methods

One of the simplest pixel-wise approaches is the maximum likelihood classification. Each class is estimated with a multivariate normal distribution (ω_k) which is described by the mean vector (μ_k) and the covariance matrix (Σ_k). In practice, these distribution parameters can be estimated using the

[3]Distriduted by EURIMAGE S.c.r.l., © ESA 2003.

[4]Original data provided by EUROMAP GmbH, © EOSAT, ANTRIX 2004.

training reference map. If $x_1, x_2, \ldots, x_{n_k}$ vectors belong to the class k in the reference map, then the class descriptors can be calculated as follows:

$$\mu_k = \frac{\sum_{i=1}^{n_k} x_i}{n_k} \tag{2.1}$$

$$\Sigma_k = \frac{n_k \sum_{i=1}^{n_k} x_i x_i^{\mathrm{T}} - (\sum_{i=1}^{n_k} x_i)(\sum_{i=1}^{n_k} x_i)^{\mathrm{T}}}{n_k(n_k-1)} \tag{2.2}$$

This expression determines the density of an x vector in distribution ω_k:

$$\mathrm{P}(x|\omega_k) = (2\pi)^{-N/2} |\Sigma_k|^{-1/2} e^{(x-\mu_k)^{\mathrm{T}} \Sigma_k^{-1} (x-\mu_k)/2} \tag{2.3}$$

Beside the class distribution, a prior probability $P(\omega_k)$ is assigned to each class. The maximum likelihood method maximizes the pixel density over the classes:

$$\max_k \mathrm{P}(x|\omega_k)\mathrm{P}(\omega_k) \tag{2.4}$$

The same optimization problem can be also derived, if we want to minimize the expected value of the total classification error:

$$\min_k \mathrm{E}(\varepsilon_k|x) = \sum_{i=1}^{n} \chi(k \neq i)\mathrm{P}(\omega_k|x) \tag{2.5}$$

Using the Bayes-theorem and the Law of total probability, we get the following formula:

$$\min_k \mathrm{E}(\varepsilon_k|x) = \frac{\sum_{i=1}^{n} \chi(k \neq i)\mathrm{P}(x|\omega_k)\mathrm{P}(\omega_k)}{\sum_{i=1}^{n} \mathrm{P}(x|\omega_k)\mathrm{P}(\omega_k)}, \tag{2.6}$$

which is identical to the maximization problem (2.4). A more general approach is when the χ function is replaced with an arbitrary cost function:

$$\min_k \mathrm{E}(\varepsilon_k|x) = \frac{\sum_{i=1}^{n} \lambda_{ki}\mathrm{P}(x|\omega_k)\mathrm{P}(\omega_k)}{\sum_{i=1}^{n} \mathrm{P}(x|\omega_k)\mathrm{P}(\omega_k)}, \tag{2.7}$$

where λ_{ki} is the cost of that case when a unit area is classified to the class k, but the area belongs to class i.

2.2.2 Classification with clustering

The biggest limitation of the Bayes method is that the categories are approximated with normal distributions, therefore the class descriptions can

characterize the thematic categories inaccurately. For example, parcels of alfalfa are harvested at different times of year, which leads to multiple clusters in the spectral space.

The clustering-based classification algorithms first determine the clusters of the reflectance vectors without any assumption on the thematic categories, afterward the clusters are assigned to classes [45]. The clustering can be done with various algorithms, for example K-means or ISODATA clustering.

The K-means algorithm tries to minimize the sum of square Euclidean-distance of the reflectance vectors from the cluster centers with restricting the number of clusters in K [21]:

$$\min_{C} \sum_{i=1}^{K} \sum_{x \in C_i} \|x - c_i\|^2, \tag{2.8}$$

where c_i is the center of the i^{th} cluster.

The K-means clustering is an NP-hard problem [1], but efficient heuristics and approximation algorithms exist [2].

The Lloyd's algorithm provides a good heuristic solution with the following iterative approach:

1. Start with random cluster centers (sampling from the vectors)
2. Assign each vector to the closest cluster center
3. Recalculate the cluster centers as the mean of the corresponding vectors
4. If the centers were moved significantly, then go back to step 2

The ISODATA clustering is similar to the Lloyd's algorithm, but it handles the number of clusters as a soft constraint. Therefore, during an iteration clusters can be split, merged and erased when appropriate conditions hold.

Afterward, the created clusters have to be assigned to thematic categories, where the training reference data can be used. For each cluster, we determine the most relevant category, i.e. the category which has the most reference pixel among the cluster pixels, and all cluster pixels are assigned to this class. This assignment can be improved with heuristics [43].

2.3 Segment-based classification methods

The pixel-wise classification algorithms assumes that the pixels of the satellite image are independent, they do not consider the spatial structure of the image objects. In practice, the pixels form contiguous areas, which have uniform land cover and spectral properties.

In high resolution images, the segment-based classification can decrease the error of point-like misclassifications. The pixel-wise methods classifies some of the uncertain pixels into wrong classes, which can be partially corrected with the fine-tuning of the algorithm parameters. With segment-based methods, the surrounding environment can help to classify these pixels into the right categories.

There are applications where the usage of very high resolution satellite imagery increases significantly the classification accuracy [44], for example, in the detection of scattered trees and bushes on pasture, in the observation of red mud spill or in the ragweed monitoring. These three applications and the recognition of built infrastructure in urban areas using orthophotos was investigated in [15]. In very high resolution images, the individual pixels are not representative for current land cover, therefore in these cases the pixel-wise methods are not suitable. In this case, the segment and object-based image analysis is more adequate method.

In the presented approach, first the image is split to segments, and the created segments are classified into thematic categories. During the segmentation, we consider the spatial properties of the image which can improve the accuracy of the classification.

It should be mentioned, that there are other approaches which takes into consideration the spatial structure directly in the classification process (for example, see [29]).

2.3.1 Maximum likelihood method

The pixel-wise maximum likelihood method can be generalized to classify segments. The density of that each pixel of a segment belongs to a class can be computed as the product of the pixel-wise densities.

$$\mathrm{P}(S|\omega_k) = \prod_{x \in S} \mathrm{P}(x|\omega_k) \tag{2.9}$$

The logarithm of the density can be computed from the estimated mean vector and covariance matrix of the segment:

$$\begin{aligned} &\log \mathrm{P}(S|\omega_k) \qquad\qquad (2.10)\\ &= \frac{\sum_{x \in S} -\frac{N}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma_k| - (x-\mu_k)^{\mathrm{T}}\Sigma_k^{-1}(x-\mu_k)/2}{n_S} \\ &= -\frac{N\log(2\pi) + \log|\Sigma_k| + \mathrm{tr}(\Sigma_k^{-1}(\Sigma_S + \mu_S\mu_S^{\mathrm{T}})) - 2\mu_k^{\mathrm{T}}\Sigma_k^{-1}\mu_S + \mu_k^{\mathrm{T}}\Sigma_k^{-1}\mu_k}{2} \end{aligned}$$

We assume that the segments are homogeneous, and each pixel in the segment belongs to the same category. Therefore, we can maximize the

likelihood of the segment within the classes:

$$\max_k \mathrm{P}(S|\omega_k)\mathrm{P}(\omega_k), \tag{2.11}$$

and assign each pixel in the segment to the class with the maximum likelihood.

2.3.2 Classification by spectral similarities

If the segments are large enough, then their spectral properties are similar to the spectral properties of the whole class. This allows to make a classification which checks not only that the segment distribution matches the class distribution, but also that the class distribution matches the segment distribution.

There are several distance functions, which measure the spectral similarity (or literally, the dissimilarity) of two distributions. In this study, the Bhattacharyya-distance function was used to determine the most similar class to a segment:

$$\begin{aligned} D(\omega_S, \omega_C) &= -\log \int \sqrt{f_{\omega_S}(x) f_{\omega_C}(x)} dx \qquad (2.12) \\ &= \frac{1}{8}(\mu_S - \mu_C)^{\mathrm{T}} \left(\frac{\Sigma_S + \Sigma_C}{2} \right)^{-1} (\mu_S - \mu_C) + \frac{1}{2} \log \left(\frac{\left| \frac{\Sigma_S + \Sigma_C}{2} \right|}{\sqrt{|\Sigma_S||\Sigma_C|}} \right) \end{aligned}$$

If the matrix Σ_S is singular, then the Bhattacharyya distance cannot be calculated. In practice, either the covariance matrix can be perturbed or we can formally eliminate $|\Sigma_S|$ from the expression. This can be done because it increases the distance to every class identically.

Similarly to the maximum likelihood method, we assign each pixel of a segment to that class, which minimizes the distance.

2.3.3 Classification with segment clustering

This method is also a generalization of the pixel-wise algorithm. Compared to the K-means clustering the objective function is not changed, but we add an additional constraint, that each pixel of a segment must be assigned to the same cluster.

$$\min_C \sum_{i=1}^{K} \sum_{S \in C_i} \sum_{x \in S} \|x - c_i\|^2 \tag{2.13}$$

The sum of the Euclidean-distance squares from the segment pixels to the center of cluster C_i can be calculated with the following expression:

$$\sum_{x \in S} \|x - c_i\|^2 = n_S \left(\|\mu_S - c_i\|^2 + \mathrm{tr}\Sigma_S \right) \tag{2.14}$$

Because the $n_S \mathrm{tr}\Sigma_S$ part is independent from the current cluster, therefore the clustering problem is simplified to the clustering of the segment centers weighted by the cluster sizes.

The Lloyd's algorithm can be modified to take into consideration the segment sizes.

1. Start with random cluster centers (sampling from the segment centers)
2. Assign each segment center to the closest cluster center
3. Recalculate the cluster centers as the weighted average of the corresponding segment centers

$$\mu_C = \frac{\sum_{S \in C} n_S \mu_S}{\sum_{S \in C} n_S} \tag{2.15}$$

4. If the centers were moved significantly, then go back to step 2

Better clustering can be achieved if we describe the clusters with distributions. Like in the section 2.3.2, we can assume that the segments and the corresponding clusters are spectrally similar, therefore we can use a distance function, which measures the similarity between the distributions. This leads to the following algorithm:

1. Start with random clusters (sampling from the segment distributions)
2. Assign each segment to the closest cluster using a similarity measure (for example with Bhattacharyya-distance)
3. Recalculate the clusters

$$n_C = \sum_{S \in C} n_S \tag{2.16}$$

$$\mu_C = \frac{\sum_{S \in C} n_S \mu_S}{n_C} \tag{2.17}$$

$$\Sigma_C = \frac{\sum_{S \in C} n_S \left(\Sigma_S + \mu_S \mu_S^{\mathrm{T}} \right) - n_C \mu_C \mu_C^{\mathrm{T}}}{n_C} \tag{2.18}$$

4. If the clusters were moved significantly, then go back to the step 2

This algorithm is more robust, because it can distinguish between segments with similar mean vectors but different covariance matrices.

2.4 Segmentation methods

There are numerous methods in the image processing for the image segmentation problem. In this section four graph-based methods are described.

In image processing algorithms, it is natural to represent the image as a graph. A $G(V, E)$ undirected graph can be assigned to an image, where nodes belong to pixels, and edges connect nodes belonging to neighboring pixels. Thus, a grid graph is obtained. The graph belonging to a $W \times H$ sized image has $n = WH$ nodes and $m = W(H-1) + (W-1)H$ edges. As a consequence, the size of graph is linear in the size of image.

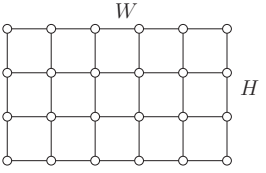


Figure 2.3: The grid graph

Each edge $(u, v) \in E$ has a weight $w : E \to \mathbf{R}_0^+$, which is a measure of dissimilarity between the adjacent pixels. Several methods can be used to calculate the difference between pixel intensities. In this study, the transformed Mahalanobis distance function is used, where Σ is the estimated covariance matrix of pixel intensities.

$$d(x_1, x_2) = e^{-(x_1-x_2)^T \Sigma^{-1} (x_1-x_2)} \tag{2.19}$$

Although some of the algorithms allow the use of extended neighborhood system, all of the studied image segmentation methods in this evaluation use the grid graph representation only with the (2.19) similarity measure.

In the terms of graph theory, segments are connected subgraphs. The aim of image segmentation is to partition the nodes into a complete disjoint system, where each partition is connected.

In the implementation of the following algorithms, the LEMON library was extensively used (see Section 5).

2.4.1 Best merge segmentation

Best merge is a greedy approach for region merging image segmentation. In each iterative step the algorithm contracts those two adjacent segments that are most tightly connected.

In the course of the algorithm, the state of segmentation is represented as a graph. A node is assigned to each segment, and nodes belonging to adjacent segments are connected. The similarity between segments is measured with the average weight of the edges of common boundary.

The algorithm first constructs a grid graph and calculates initial weights. Moreover, initially every pixel forms a separate segment. In the following steps, it chooses the edge with maximum weight, and contracts its two end nodes. At the level of image, this means that the segments on the two sides of this edge are merged. If the contracted nodes have common neighbors, then only one edge is kept and its weight is recalculated.

This algorithm was described by Beaulieu and Goldberg [3] and by Tilton [67], using different similarity measures. The authors of this article use average connection weight, as it can be determined from the edge weights of initial graph.

Two data structures are used to implement an efficient algorithm. The edges with their weight are stored in binary heap [66]. It can be initialized in $O(n)$ time, the minimum weight edge can be obtained in $O(1)$ time and after a merge, an edge weight can be recalculated in $O(\log(n))$. Pixels are stored in a union-find data structure [66], hence two segments can be merged in $O(\alpha(m, n))$, where α is the inverse Ackermann function.

Generally, the best merge algorithm has quadratic worst case running time, because edge weights have to be recalculated after every region merging, and its running time can be proportional to the degrees of corresponding nodes. In practice, merges are often very efficient since the average degree of nodes in a planar graph is always less than six.

Claim 1 *Best merge algorithm with average edge weight distance can be implemented in* $O(n \log^2(n))$ *worst case running time.*

We have to limit the number of steps during merges. The incident edges can be stored in a self balancing binary search tree for each node, therefore we can determine whether there is an edge connecting two given nodes in $O(\log(n))$. If the smaller segment is merged to the bigger one in every phase, we can achieve the appropriate time bound. Instead of the binary search trees, hash tables can be used, as well.

2.4.2 Tree merge segmentation

Tree merge algorithm is described by Felzenszwalb and Huttenlocher [24]. The algorithm's aim is to provide an efficient, greedy decision-based segmentation method to handle better the local variability of the image.

Initially, each pixel belongs to a separate segment. Later, in each step, adjacent segments are merged, as in the best merge algorithm. Edges are taken in descending order by weight, and it is decided whether the two segments belonging to the two end nodes can be contracted. If the two end nodes are already in the same segment, then there is nothing to do. Else the internal state of segments is compared to their difference.

Let the internal variability of a segment be the minimum weight of edges in the maximum weight spanning tree (MWST) of its subgraph:

$$\mathrm{Int}(S) = \min_{e \in \mathrm{MWST}(S,w)} w_e \tag{2.20}$$

The difference between two components is measured as the maximum weight edge that connects them:

$$\mathrm{Diff}(S_1, S_2) = \max_{e(u,v), u \in S_1, v \in S_2} w_e \tag{2.21}$$

If the difference between two segments is not much less than the internal variability of either segment, then they are merged:

$$\mathrm{Diff}(S_1, S_2) \geq \min(\mathrm{Int}(S_1) - \tau(S_1), \mathrm{Int}(S_2) - \tau(S_2)), \tag{2.22}$$

where $\tau(S)$ is a threshold function, defined as $\tau(S) = \frac{C}{|V(S)|}$ in this experiments, with an appropriate C constant.

This algorithm can be implemented efficiently with union-find data structure [66]. The overall running time is $O(n \log(n))$, which is determined by the initial sorting of edges.

2.4.3 Minimum mean cut segmentation

Minimum mean cut-based segmentation [71] is a hierarchical process, recursively splitting the image into segments. In every phase we split up a segment so that the mean cut between two subregions is minimized.

$$C_m(X) = \frac{\sum_{e(u,v) \in E, u \in X, v \in V \setminus X} w_{uv}}{\sum_{e(u,v) \in E, u \in X, v \in V \setminus X} 1} \tag{2.23}$$

The algorithm has a generalized variant, which splits the regions by the *minimum ratio cut* [72]. If weight function $\hat{w}$ is identically 1, the method falls back to minimum mean cut algorithm.

$$C_r(X) = \frac{\sum_{e(u,v) \in E, u \in X, v \in V \setminus X} w_{uv}}{\sum_{e(u,v) \in E, u \in X, v \in V \setminus X} \hat{w}_{uv}} \tag{2.24}$$

Although the minimum ratio cut problem approximation is NP-hard in general graphs, and for minimum mean cut problem there is no known polynomial time algorithm, both problems can be solved in polynomial time for planar graphs and, as a consequence, for grid graphs. The minimum mean cut approximation, which is used in this experimental study, can be reduced to weighted perfect matching in four steps:

1. The planar minimum mean cut problem can be reduced to finding a minimum mean cycle in an undirected graph. We use the dual $\hat{G} = (\hat{V}, \hat{E})$ of the planar graph $G(V, E)$. G is naturally embedded into the plane by the image, i.e. pixel coordinates determine the coordinates of the nodes, and nodes can be connected by straight lines. The nodes in $\hat{G}$ are assigned to each region surrounded by edges in G. Moreover, dual edges are mapped one-to-one to the primal edges: if a primal edge e connects the nodes u and v, then dual edge $\hat{e}$ connects the dual nodes $\hat{u}$ and $\hat{v}$ belonging to the regions beside e.

   The cuts in primal graphs can be transformed to a set of cycles in dual graph and if the cut is connected, we get only one dual cycle. Finding the minimum mean cut in G with weights w_e is equivalent to finding the minimum mean cycle in $\hat{G}$ with $w_{\hat{e}} = w_e$.

2. The ϵ-approximation of minimum mean cycle problem can be transformed to finding a negative cycle in an undirected graph. Let $G = (V, E)$ be a graph with w_e weight function. The graph contains negative cycle with respect to $\hat{w}_e = w_e - \lambda$ weights, if and only if the optimum value of the minimum mean cycle is less than λ.

   Similarly, minimum ratio cut problem has optimum less or equal to λ if and only if the graph has negative cycle with respect to $w_e - \lambda \hat{w}_e$.

   If there exist a λ_0 and λ_1 lower and upper bounds for λ, then the optimum value can be approximated with binary search. In the original papers ([71], [72]), $\lambda_0 = \min_{e \in E} w_e$ and $\lambda_1 = \max_{e \in E} w_e$ are chosen for the minimum mean cut algorithm and $\lambda_0 = \min_{e \in E} \frac{w_e}{\hat{w}_e}$ and $\lambda_1 = \max_{e \in E} \frac{w_e}{\hat{w}_e}$ are bounds for the minimum ratio cut problem.

3. The negative cycle problem can be reduced to minimum weight perfect matching problem. Let $G = (V, E)$ be an undirected graph and w_e weight function. We transform G to $\hat{G} = (\hat{V}, \hat{E})$ graph (see Fig. 2.4). If $u \in V$, then we add $u_1, u_2 \in \hat{V}$ nodes to $\hat{G}$ and we connect them with an edge. If $e(u, v) \in E$, we add $u_e, v_e \in \hat{V}$ to $\hat{G}$ and connect them. We connect (u_1, u_e), (u_2, u_e), (v_1, v_e), (v_2, v_e). We construct a $\hat{w}_{\hat{e}}$ weight function: $\hat{w}_{u_1 u_2} = \hat{w}_{u_e, v_e} = 0$ and $\hat{w}_{u_1 u_e} = \hat{w}_{u_2 u_e} = \hat{w}_{v_1 v_e} = \hat{w}_{v_2 v_e} = \frac{w_e}{2}$.

In $\hat{G}$ the weight of minimum perfect matching is always non-positive,

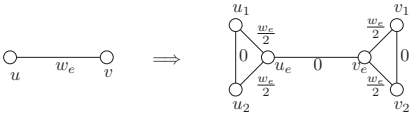


Figure 2.4: Transformation for computing a negative cycle

and if the original graph G has a negative cycle, then the transformed graph $\hat{G}$ has a negative perfect matching with the same weight.

4. The minimum perfect matching can be solved with Edmond's algorithm. The straightforward implementation has $O(n^2m)$ running time, where n and m are the number of nodes and edges, respectively. Time complexity can be improved by using advanced data structures. The running time of the best known algorithm for the minimum perfect matching problem is $O(n^2 \log(n)+nm)$ [26]. Beside theoretical bounds, the practical implementations have also been improved [12, 52, 41].

The total running time is $O\left(n^2 \log(n) \log \frac{\max_{e\in E} w_e}{\epsilon \min_{e\in E} w_e}\right)$ for minimum mean cut and $O\left(n^2 \log(n) \log\left(\max_{e\in E} \frac{w_e}{\hat{w}_e} / \epsilon \min_{e\in E} \frac{w_e}{\hat{w}_e}\right)\right)$ for minimum ratio cut algorithm. Therefore, these methods yield *polynomial time approximation scheme* (PTAS).

Claim 2 *Both the minimum mean cut and minimum ratio cut algorithms can be solved in strongly polynomial time.*

Step 2 can be replaced by the following algorithm. Starting from $\lambda = \lambda_1$, we obtain a solution C of the minimum cycle problem. Based on the cycle system found, we can update $\lambda = \frac{w_C}{\hat{w}_C}$. For the minimum mean cut problem, the number of edges in consequent solutions is decreased at least by one, therefore the optimal solution can be found in $O(n)$ time. This solution yields $O(n(n^2 \log(n) + nm))$ running time.

The minimum ratio cut problem can also be solved in strongly polynomial time. The method of Norton et al. [53] yields a running time of $O\left((n^2 \log(n) + nm)^2\right)$.

The minimum ratio cut segmentation was implemented with the binary search-based approximation algorithm. For internal edges, we use the distance function defined in (2.19) for w and identically 1 for $\hat{w}$. Both w and $\hat{w}$ have the value 0 for external edges. The usage of this weight functions can

also be interpreted as using the minimum mean cut segmentation, but the edges on external faces are contracted.

In the investigated implementation, running time is improved with the minimum spanning tree heuristic. Because the ratio of the maximum and minimum edge weight can be large, too many iterations may be needed. To avoid this, let us take the minimum weight edge outside the minimum spanning tree of the graph. First, the minimum spanning tree of the graph is calculated. Let w^* be the minimal weight of edges outside the minimum spanning tree. With this choice, $\lambda_0 = \frac{w^*}{n}$ is a lower bound and $\lambda_1 = w^*$ is an upper bound on the minimum mean cycle.

An $O(nm \log(n))$ running time minimum weighted matching algorithm is used, which is implemented in the Library for Efficient Modeling and Optimization in Networks [47].

2.4.4 Normalized cut segmentation

As in the case of the minimum mean cut method, *normalized cut algorithm* [60] also starts with one segment containing the whole image and splits it hierarchically into subregions in later steps. In each step it searches for the cut that minimizes normalized cut, which is defined as follows:

$$C_n(X) = \frac{\sum_{u \in X, v \in V \setminus X} w_{uv}}{\sum_{u \in X, v \in V} w_{uv}} + \frac{\sum_{u \in X, v \in V \setminus X} w_{uv}}{\sum_{u \in V \setminus X, v \in V} w_{uv}} \tag{2.25}$$

Unfortunately, the minimum normalized cut problem is NP-hard even for grid graphs, therefore an efficient exact algorithm cannot be expected. The best known polynomial time approximation for this problem is $O(\log(n))$ [46], and there exists a constant-factor approximation for planar graphs [55]. For the image segmentation problem, the original paper [60] suggests a heuristic approximation algorithm based on eigenvector computation.

The original problem can be formulated in matrix form, with the following notation: $V = \{1, 2, \ldots, n\}$ and $W_{ij} = w(i, j)$. Furthermore, let $d_i = \sum_{j \in V} w_{ij}$ vector and $D_{ii} = d_i$ diagonal matrix. We are looking for an $x : V \to \{1, -\alpha\}$ indicator vector (where $\alpha = \frac{\sum_{x_i > 0} d_i}{\sum_{x_i < 0} d_i}$) that minimizes the following quadratic form:

$$\min \frac{x^{\mathrm{T}}(D - W)x}{x^{\mathrm{T}}Dx} \tag{2.26}$$

We can relax this problem by dropping the restricted value set, i.e. the objective value is optimized on $x : V \to \mathbb{R}$ value set, and the relaxed problem can be reformulated as an eigenvector problem. Especially, the eigenvector

of the normalized *Laplacian* with the second smallest magnitude has to be found:

$$D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}x = \lambda x \tag{2.27}$$

In image processing algorithms, images are usually interpreted as grid graphs. This method allows the use of any extended neighborhood system; for example, in the original paper nodes are connected if their spatial distance is less than a threshold. In the current implementation grid graph neighborhood system is used.

Because grid graphs are bipartite graphs as well, the following statements are valid for its normalized Laplacian $D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}$ [9]:

- Let B be a diagonal matrix, where $B_{ii} = 1$ for one side of the bipartite graph and -1 for the other side. If the graph has x eigenvector with λ eigenvalue, Bx is also an eigenvector with $2-\lambda$ eigenvalue.
- All eigenvalues are in the range $[0, 2]$.
- The vector $D^{\frac{1}{2}}\mathbf{1}$ is an eigenvector with eigenvalue of 0 and $D^{\frac{1}{2}}B\mathbf{1}$ is an eigenvector with eigenvalue of 2.

Applying these properties, we can reduce the relaxed problem to finding the largest eigenvector of the normalized Laplacian, which is perpendicular to $D^{\frac{1}{2}}B\mathbf{1}$. We use power method for this task:

$x \leftarrow \mathcal{N}(\mathbf{0}, I)$
while λ **is not converged do**
  $x' \leftarrow D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}x$
  $x' \leftarrow x' - \frac{(D^{1/2}B\mathbf{1})^T x' D^{1/2}B\mathbf{1}}{(D^{1/2}B\mathbf{1})^T D^{1/2}B\mathbf{1}}$
  $\lambda \leftarrow x'^T x$
  $x \leftarrow x'/\|x'\|$
end while

There are several possibilities to derive a partitioning from the relaxed solution. In the original paper, nodes are sorted by the corresponding components of eigenvector, and best cut value is chosen from K evenly spaced splittings. This method was improved, and the normalized cut on all possible splitting points was computed in linear time.

$C \leftarrow 0, A_X \leftarrow 0, A_{V\setminus X} \leftarrow \sum_{e\in E} w_e, X \leftarrow \emptyset$
for $u \leftarrow$ **nodes order by value do**
  $X \leftarrow X \cup \{u\}$
  for $e(u,v) \in \mathrm{E}_{\mathrm{inc}}(u)$ **do**
    $A_X \leftarrow A_X + w_e$, $A_{V\setminus X} \leftarrow A_{V\setminus X} - w_e$
    if $v \in X$ **then** $C \leftarrow C - w_e$ **else** $C \leftarrow C + w_e$ **end if**

end for
if $\frac{C}{A_X} + \frac{C}{A_{V \setminus X}}$ **improves best cut then**
update best cut
end if
end for

Since edge weights are stored as floating point numbers, we have to deal with inexact numerical computation, and to refine the previous algorithm. We use this method only to calculate cut values for the first half of the sequence; the second half is determined in backward order. In addition, cut values are recalculated if their inaccuracy becomes potentially large.

2.5 Experimental results

In [16], we compared and evaluated the four graph-based segmentation algorithms described in section 2.4 with maximum likelihood and spectral similarity-based classification methods. In previous research [43], we have already shown that the segment-based classification can overtake the pixel-wise algorithms.

It can be stated that there is no general "best segmentation". What we can do is to try to find a good algorithm for a given domain (the set of thematic classes, the area where the procedure should be applied, the kind of input data etc.) and with respect to some goals (thematic accuracy, visual suitability, possibilities and limitations of parametrization).

Classification accuracy assessment is the most important aspect of comparison. It is measured on the basis of a test reference image. We use the overall classification error, the error matrix and the error map to evaluate accuracy.

All of the four segmentation algorithms depend on control parameters. With each method seven different segmentations are generated with different parameter settings. The author's experiments show that there is no uniformly good parametrization for an algorithm, the best option depends on actual satellite images and training data. Apparently, the determination of a good parameter value multiplies the computing effort, but except the tree merge algorithm the segmentations for different parameters can be computed in one phase without significant increment of running time. In the following part of the subsection, the best of the seven results are used for each segmentation algorithm.

In real production systems, the parametrization of the algorithms is supported by human expertise and knowledge. Moreover, the human expert can also override the generated result of the classification by additional ground

#	Pixel-wise	Best merge		Tree merge		Norm. cut		Mean cut	
		Bhat.	ML	Bhat.	ML	Bhat.	ML	Bhat.	ML
1	91.76	96.17	95.37	89.96	92.07	94.18	85.66	94.89	94.25
2	95.15	87.06	84.32	90.43	90.58	91.62	85.44	93.21	86.70
3	92.32	90.79	88.78	94.10	91.03	96.59	93.32	93.23	90.22
4	23.61	74.33	54.05	82.82	53.74	88.64	49.72	76.36	49.24
5	89.69	90.31	86.50	92.41	88.11	92.59	88.45	92.68	94.10
6	85.65	84.29	87.44	87.39	90.66	90.24	92.49	88.54	90.45
7	65.53	85.32	22.57	89.54	24.21	92.38	24.06	81.89	24.03
8	94.31	89.97	86.31	95.84	94.94	96.70	95.68	95.92	93.35
9	77.97	65.44	72.71	72.18	77.70	76.42	83.25	70.58	78.08
10	95.40	93.55	93.44	96.39	95.46	97.37	95.71	96.89	96.91
AVG	81.14	85.72	77.15	89.11	79.85	91.67	79.38	88.41	79.73

Table 2.1: Classification results

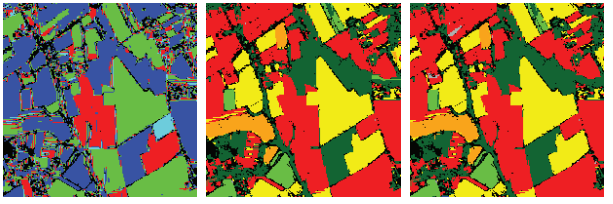
data or by own experience. For the sake of unbiased results, additional post-processing step was not used in this study.

The classification method has also influence on the accuracy. Each segmented image was classified both with the Bayes algorithm and the Bhattacharyya-distance method. None of them outperformed the other clearly, but Bhattacharyya-distance gave better results by 9.7% points in average and in the 73.3% of the test cases it gave better overall accuracy.

Both best merge and tree merge algorithms perform well in classification. *Tree merge* yielded better overall accuracy in 9 of the 10 test cases than *best merge*. Moreover, the running time of *tree merge* algorithm is significantly shorter. Best merge delimits segments of very similar size: the deviation of segment size is the smallest among the four algorithms. The segments resulted from *tree merge* show much larger variety in size. There seem to be many small noise-like spots in the segmentation results; however, their presence can be explained by the differences in soil properties and crop development. In addition, a lot of these spots disappear in the final classification result (class map).

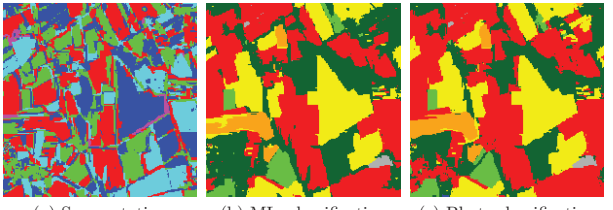
Out of the four segmentation algorithms discussed, *minimum mean cut segmentation* has the longest running time, which can be explained by two reasons. First, finding the minimum mean cut in undirected graphs use the weighted perfect matching algorithm on huge graphs, which is a costly operation. The second reason is that the splitting of segments is very unbalanced, i.e. just a few pixels (actually, in many cases, only one pixel) are detached from big segments. This phenomenon usually yields bad performance in "divide and conquer" algorithms. To speed up the algorithm, we have initially split the whole image into equal sized tiles, but even with relatively small tiles this algorithm still remains the slowest of all.

Unbalanced splits have an impact on image segmentation quality as well.



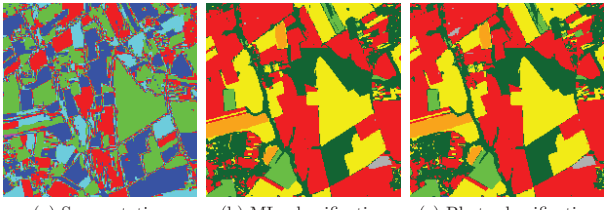
(a) Segmentation (b) ML. classification (c) Bhat. classification

Figure 2.5: Best merge segmentation



(a) Segmentation (b) ML. classification (c) Bhat. classification

Figure 2.6: Tree merge segmentation



(a) Segmentation (b) ML. classification (c) Bhat. classification

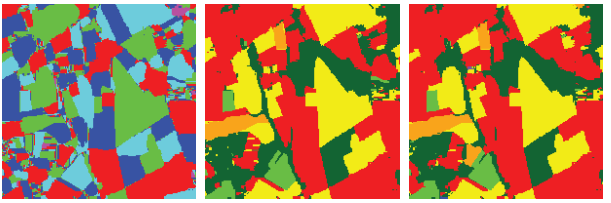
Figure 2.7: Minimum mean cut segmentation

If there is a smooth boundary between different land cover categories, this algorithm tends to delimit individual pixels as new segments. An artificial image and the state of image segmentation after various stages is seen in Fig. 2.8. Individual pixels do not separate the black and white regions; they appear sporadically in a wide belt along the transition zone, and they prevent the algorithm from finding a good separator cut. Coming back to the crop mapping application, this "salt and pepper noise" remains spectacular in the final class map. High deviation in segment size is caused by this phenomenon and by the fact that this algorithm highly depends on the local variety of pixel intensities.

(a) Image (b) 1^{st} (c) 10^{th} (d) 30^{th} (e) 39^{th}

Figure 2.8: Minimum mean cut segmentation steps on artificial image

In these experiments, the *top-down* algorithms performed slightly better than the *bottom-up* methods; and the *normalized cut* algorithm gave overall the best results. The variety in segment size is relatively high, but there are not many noise-like spots or individual pixels (compared to tree merge and minimum mean cut segmentation). Out of the four algorithms, minimum normalized cut algorithm yields results that are *visually* the most favorable, the most realistic: segment boundaries align the best to parcel boundaries observed in the field.



(a) Segmentation (b) ML. classification (c) Bhat. classification

Figure 2.9: Minimum normalized cut segmentation

It is also worth investigating the individual input sets. In the cases 4 and 7

the pixel-wise classification performed poorly. In both of these cases there was a category "other classified crops" which has large spectral variability. Pixel-wise classification found too many pixels matching to this category (see Table 2.2), which dramatically decreased the overall accuracy. The segmentation-based methods, especially with the Bhattacharyya classification, can handle the classes with large deviation (Table 2.3). In other sets, similar problems occur with the category "pasture and grass".

	C1	C3	C4	C10	C12	C18	C26	C27	C30	C55
C1	1397	0	0	0	0	522	0	5	3	466
C3	67	0	0	0	0	15	0	0	1	187
C4	2	10	0	0	0	2	0	0	0	87
C10	0	0	0	12278	28	0	4	17	0	5220
C12	0	0	0	463	550	0	0	0	0	402
C18	0	0	0	0	0	3334	0	0	0	34
C26	0	0	0	0	0	0	180	6	0	394
C27	0	0	0	0	0	0	5	14	1	1146
C30	3	0	0	0	0	0	0	0	5	248
C55	0	0	0	0	0	0	0	0	0	0

Table 2.2: Pixel-wise maximum likelihood classification

	C1	C3	C4	C10	C12	C18	C26	C27	C30	C55
C1	1679	104	0	7	0	585	1	0	0	17
C3	115	78	0	6	0	2	0	50	0	19
C4	0	99	0	1	0	0	0	0	0	1
C10	1	3	0	17265	0	13	0	89	11	165
C12	0	0	0	550	855	0	0	5	0	5
C18	1	0	0	11	0	3285	0	16	0	55
C26	0	0	0	20	0	0	496	29	0	35
C27	0	0	0	29	0	0	0	1128	0	9
C30	5	4	0	0	0	0	0	0	246	1
C55	0	0	0	0	0	0	0	0	0	0

Table 2.3: Normalized cut segmentation with Bhattacharyya-distance classification

2.6 Conclusion

We can summarize the results of the research in the following points:

- A framework has been created for solving and evaluating classification problems in remote sensing, and several segmentation, clustering and classification algorithms have been implemented in this framework. The graph-based segmentation algorithms were investigated advantageously. Therefore, the best merge segmentation was interpreted as graph-based algorithm, and the normalized cut segmentation was simplified to use the power method.

- According to the research, the segment-based classification algorithms can outperform the pixel-wise methods both in accuracy and runtime efficiency.

- The experiments show that the top-down segmentation methods have better thematic accuracy than the bottom-up algorithms, since they are less affected by the differences in the spectral deviation of the classes.

- The segment-based classification methods with Bhattacharyya-distance decision rule are more robust against classification errors than with the maximum likelihood decision rule, since they handle more accurately the segments with large spectral extent.

- The minimum normalized cut image segmentation with Bhattacharyya-distance classification algorithm provides a robust and efficient method for thematic classification based on an exhaustive test.

3 Raster-vector conversion of maps

The raster-vector conversion of maps aims at creating digital maps which can be used in spatial databases and vector-based GIS systems. In this section, the IRIS project and its optimization related tasks are presented[50, 19].

The vector maps have numerous advantages in contrast to raster maps. Namely, the objects are clearly identifiable in the vector map, the resolution can be chosen arbitrarily, geometric queries about the maps can be computed efficiently, and finally, the size of the vector maps are considerably smaller than the raster maps.

The governments, authorities and service providers tend to use topographic maps in vector format [65]. The vectorization of existing raster maps is an important and serious challenge. The manual process is inefficient both in time and costs, therefore the demand for an automatic conversion tool is high.

There are different levels of supporting the raster-vector conversion with computers. For example, it is already a significant improvement of the process, if a software visually enhances the relevant information in the map. The process can be improved, if the software automatically recognizes various objects in the image. Each improvement step in the automatizing needs more algorithmic solution.

In the presented framework, which was developed under the IRIS project, the process of the map conversion is split into three distinct steps: preprocessing, vector retrieval and postprocessing. In the preprocessing step, the raster image is simplified in order to make the maps more suitable for the vectorization algorithms. In the second step, the vector data are retrieved from the preprocessed raster. Finally, in the postprocessing the quality of the vector image is improved.

The map objects are classified by their properties into three levels: fields, curves and symbols. In the first two steps, the different kind of objects are processed separately, while the object types are used together to improve the result of the vector image in the last step.

The author was working with the leading of István Elek and together with Zsigmond Máriás on the raster-vector conversion problems. His main tasks were to develop the vectorization algorithms for already preprocessed images and implement methods for the postprocessing. He was also working on fine tuning of the preprocessing steps, and providing suitable workflows for Hungarian topographic maps. Based on these results, Rudolf Szendrei continued on working raster-vector conversion problems.

A complete theoretical workflow for topographic map processing is described by Levachkine [48]. The partitioning of his conversion method is

similar to the currently presented solution, however the details of the applied methods are quite different.

Several special subproblems are raised in the map vectorization process. Contour line recognition was described by Khotanzad and Zink [37], and Salvatore and Guitton [58]. Text processing problems on cartographic maps were studied by Cao and Tan [7], and Velázquez and Levachkine [69], and road map extraction was discussed by Wang and Bao [70], and Chiang, Knoblock and Chen [39].

3.1 Features of topographic maps

The main goal of the IRIS project was to convert scanned Hungarian topographic maps to vector maps. The maps of each country (as the Hungarian ones) may have special features, however the described methods are rather general, and they are able to process, after some customization, various kinds of maps. The specification of Hungarian topographic maps is well-defined, but it contains several complex rules. This is why a wide range of sophisticated image processing methods need to be used in the conversion.

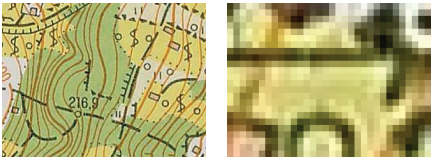


Figure 3.1: Smooth colors on scanned maps

The printing and scanning methods make the conversion process more difficult, e.g. objects, which are ideally identical, often get different colors on the printed map (see Fig. 3.1), or on the boundaries of different fields mixed colors can evolve. Moreover, it should also be considered that some of the scanned topographic maps were formerly damaged by heavy usage.

3.2 Preprocessing

The main goal of the preprocessing step is to create raster maps, so that boundary and edge detection algorithms and image matching methods can be applied.

In the preprocessing, simple image manipulation algorithms are used, e.g. digital filters, color space transformations, etc. The preprocessing is implemented as a sequence of these manipulation steps.

During the preprocessing, it can be also used, that the properties of the input image are known. The object colors, the object types and the object properties are determined by the map type, what can be used to adapt the preprocessing algorithms. For example, in a Hungarian topographic maps, the black and brown colors are never used for field objects, therefore we can try to eliminate these pixels, before the field objects are extracted from the image. Furthermore, the possible colors of the field objects are also defined, therefore supervised classification can be used to classify the pixels into field categories.

The input for the different vectorization tasks cannot be produced in a common preprocessing step. In order to identify the lines and symbols of a map, these objects must be emphasized in the image. At the same time, the appropriate processing of fields would require to remove lines and symbols. Hence, different preprocessing steps are used for different layers.

3.2.1 Digital image filters

The digital filters [74] are playing a crucial role in the conversion project. They are simple image processing procedures, which apply changes on the pictures based on local image information. A sliding window is moved over each pixel of the image, and depending on the content under the window, the new pixel value is calculated for the center of the window:

$$I'[x, y] = F(I[x - k \dots x + k, y - k \dots y + k]) \tag{3.1}$$

The running time of this algorithm is $O(WHT_{F,k,l})$, where W and H are the sizes of the image, $2k + 1$ and $2l + 1$ are the sizes of the kernel and $T_{F,k,l}$ is the time of the evaluation of F on a sub-image with $(2k + 1, 2l + 1)$ size.

Convolution filters. The filter is called convolution filter, if F is calculated as the sum of the component-wise products of a given matrix K and the sub-image under the window. The matrix K is denoted the kernel of the filter:

$$I'[x, y] = \sum_{i=-k}^{k} \sum_{j=-l}^{l} K[k + i, l + j]I[x + i, y + j] \tag{3.2}$$

The convolution filters are formulated in the spatial domain of the image, but they can be also transformed to the frequency domain. If both the image and the kernel matrix are Fourier transformed, then the convolution

is simplified to a vector product. Therefore, all frequency filters, e.g. low- or high-pass filters can be approximated with an appropriately chosen kernel matrix.

Generally, this filter can be computed in $O(WHkl)$ time, but for special convolution filters, like separable filters, this running time can be improved. If the kernel K can be expressed as the outer product of two vectors, the filter operation can be applied separable vertically and horizontally, which yields $O(WH(k+l))$ running time.

Smoothing filters. The simplest convolution filter is the box filter, where the kernel matrix contains only a constant value:

$$K[i,j] = \frac{1}{(2k+1)(2l+1)} \tag{3.3}$$

The Gauss filter is another separable convolution filter, which is one of the most widely used smoothing filter. The kernel matrix is computed from the frequency values of the two-dimensional Gaussian distribution.

$$K[k+i, l+j] = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}} \tag{3.4}$$

Edge detection filters. The following convolution filters approximate the gradient of the intensity change horizontally and vertically:

$$K_H = \begin{pmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{pmatrix} \quad K_V = \begin{pmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{pmatrix} \tag{3.5}$$

If both of the previous filters are computed on the image, we can get a gradient vector for each pixel. If the magnitude of the vector is greater than a threshold value, then the pixel most probably belongs to an edge in the image.

The edge detection can be also implemented with one filter if the Laplace filter is used:

$$K = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{3.6}$$

This filter approximates the square of the magnitude of the gradient vector. The Laplace filter is often used together with the Gauss filter which becomes more robust against noise.

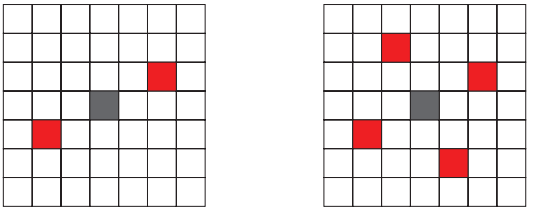
Median and rank filters. Similarly to the convolution filters, the median filter also uses a sliding window on the image, and the result of the filter is calculated as the median of the values in the window. Compared to the previously described filters, it is not linear function of the input image, therefore it cannot be expressed in the frequency domain.

A less obstructive version of the median filter is the conservative smoothing filter. It calculates the lower and upper q-quantile of the window, and if the original pixel value is not between the two calculated quantiles, then it is rounded down or up to fit into this interval. If q is fixed to 2, then we get back the median filter.

Both filters can be implemented in $O(WHkl)$ time using linear time n^{th} element selection. The algorithms can be improved to $O(WH(k+l)\log(k+l))$ with using heaps to compute the quantiles.

Edge and corner preserving filter. All smoothing filters can be modified to preserv the edges in the image with the following method. Instead of using all pixels of the filtered region, only half of them are used to compute the filter value. The pixels are handled in pairs of two oppositely located points (see Fig. 3.2a). To calculate the filter value, only that point of each pair is used which is more similar to the central point. With this modification, the filter does not smooth the boundaries and the edges on the map.

The main idea behind the corner preserving filters is similar to the edge preserving, but instead of pairs of pixels, it handles the region by four pixels (see Fig. 3.2b). The four coherent pixels are the vertexes of squares having the same center as the filter region. Analogously, only the most similar pixel to the center is used to calculate the filtered value for each possible square. This filter does not round the corners of rectangular shapes surrounded by a homogeneous field.



(a) Edge preserver filter (b) Corner preserver filter

Figure 3.2: Pixel selection for edge and corner preserver filters

Thinning filters. The thinning filters are different from the previous filters, because it might need multiple passes to achieve the final result. The aim of these filters are to determine the skeleton of the image objects without changing the topology of the objects. The algorithm works on bitmaps, i.e. each pixel value is 0 or 1.

These algorithm usually "peels" the pixels from objects while they get one pixel wide lines. We use the following notation for the environment of pixel C:

$$\begin{pmatrix} NW & N & NE \\ W & C & E \\ SW & S & SE \end{pmatrix} \tag{3.7}$$

Let us define the $t(C)$ which determines the number changes around C (in the sequence $N, NW, W, SW, S, SE, E, NE, N$). Furthermore, function $o(C)$ calculates the non-zero values around C.

The image is scanned, and if the following conditions hold for the pixel C, then it is deleted:

$$C = 1 \tag{3.8}$$
$$2 \leq o(C) \leq 6 \tag{3.9}$$
$$t(C) = 2 \tag{3.10}$$
$$N = 0 \vee W = 0 \vee E = 0 \vee t(N) \neq 2 \tag{3.11}$$
$$N = 0 \vee W = 0 \vee S = 0 \vee t(W) \neq 2 \tag{3.12}$$

This step is iterated while there are changes in the image.

Masked filter. The set of preselected pixels are regarded as a mask on the image; for example, pixels with specific colors can be selected. The mask influences the calculated filter values, e.g. in the simplest case, the selected pixels are omitted when the filter is computed. In addition, it also can determine which pixels are modified by filter, e.g. the mask pixels usually remain unchanged.

The masked filters can be interpreted in a more general view, when the pixels in the image are categorized into classes. For each class different filter functions are defined, which calculates the filter values by the intensities and the categories of the image pixels. We get back the masked filters, if we create two classes from the masked and the unmasked pixels, and we use the identity filter for the mask pixels.

The filter function also can adaptively adjust itself according to the image. For example, when a masked filter is calculated by the unmasked pixels, and the surrounding area is not enough large to get reliable result, the size of the

filter region can be increased due to get more reliable result for the newly calculated pixel value. More precisely, while the rate of the mask pixels is higher than a threshold, the filter is continuously enlarged.

Combination of filters. However, when one of the above filters is used to improve a property of the image, some other properties may weaken by this step. For example, a median filter eliminates the noises, but it also rounds down the corners of rectangular fields. Therefore, the special type of filters, like edge or corner preserving, masked or adaptive filters are combined to provide better results than a simple convolution of the same filters.

3.3 Layer of fields

The fields are marked in different colors on the topographic maps. The aim of preprocessing is generating a map with homogeneous fields with the predefined surface colors of the topographic map. A masked corner-preserving smoothing filter is used to preprocess the map. The applied mask eliminates the lines and symbols. This filter makes the fields more homogeneous without damaging the boundaries, furthermore it erases the non-surface type objects.

3.3.1 Pixel classification

The pixels of the preprocessed images are classified into color classes. The classification could be made by several methods; in this case, the maximum likelihood and simulated annealing methods [28, 29] were tested.

In Section 2.2.1, the maximum likelihood method was already presented, so it is described here only roughly. In both algorithms, the field colors are predefined, moreover samples from each surfaces are given. The intensities of the field types are estimated as multivariate normal distributions, where the μ mean vector and the Σ covariance matrix are used to describe a class. In the maximum likelihood method the pixel x is classified into the color class for which the following probability is maximum:

$$p(x|\omega) = (2\pi)^{-N/2}|\Sigma|^{-1/2}e^{-\frac{(x-\mu)^{\mathrm{T}}\Sigma^{-1}(x-\mu)}{2}} \tag{3.13}$$

Alike the maximum likelihood method, the simulated annealing algorithm uses the normal distribution mixture model, but it considers the spatial localization of the pixels, too. A cost function is defined, which stands from two components. The first part is the sum of the discriminant of the likelihood function, and it describes how the pixels fit the their distributions:

$$C_1 = \sum_x \frac{(x-\mu)^{\mathrm{T}}\Sigma^{-1}(x-\mu)}{2} \tag{3.14}$$

The second component defines the cost of the inhomogeneity in the classification. In the conversion process, constant cost is calculated for each neighbors which are classified into several groups. More sophisticated spatial cost could be also used, which considers a similarity measure between adjacent pixels.

$$C_2 = \sum_{p_x, p_y \text{ are adjacent}} \chi(c(p_x) \neq c(p_y)) C_{\text{inhomogeneity}} \tag{3.15}$$

The total cost function is minimized by the simulated annealing method. Initially, the pixels are mapped to randomly chosen classes. The classification is improved by local modifications, which are chosen with a probability that depends on the "temperature" control parameter. This control parameter is decreasing during the algorithm, hence at the beginning the modifications are chosen almost uniformly, but at the end of the algorithm increasingly the highest reductions in the cost function are chosen. (see Fig. 3.3)

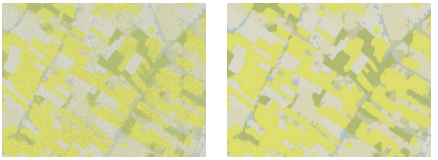


Figure 3.3: Simulated annealing in the iteration 5 and 30

The Hungarian topographic maps contain textured surfaces, which are striped mixtures of two kinds of green color. Fortunately, this problem can be solved with simple preprocessing, namely, a smoothing filter can transform this surface type into a mid-colored homogeneous field (see Fig. 3.4).

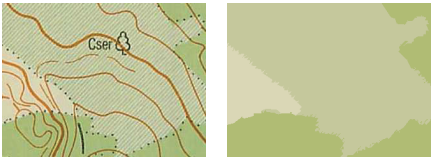


Figure 3.4: Texture recognition with smooth filters

3.3.2 Data structure for vectorized fields

The polygons are extracted from the image as the boundaries of the identically classified pixels, and they are stored in a data structure designed for consistent handling of the field layer.

The main data structure is a primal-dual graph, which stores the fields, the boundaries and the connection points of the boundaries (see Fig. 3.5). The vertexes of the primal graph are the border points, where at least three fields meet (these vertexes are marked with yellow dots in the image). A dual node belongs to each field (marked with magenta dots). The boundaries are the edges both in the primal and in the dual graphs. The edge connects two primal nodes in the graph, if there is a direct boundary between them. And it connects two dual nodes, if it is a shared boundary between the two belonging fields.

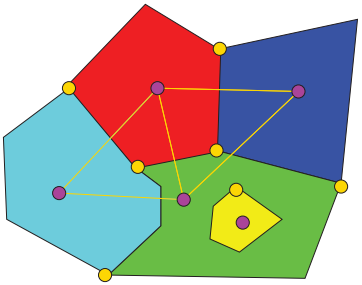


Figure 3.5: Primal-dual planar graph for the field layer

There is a secondary data structure defined on the primal-dual graph, which describes the embedding relationship between the fields. In the figure, the yellow field is a disconnected component of the graph, but we would like to assign it to the green field, because they are in embedding relation. The secondary data structure is tree, which contains the main graph component as the root node. To each component node, we add all fields inside the component as child nodes, and to each field, which contains embedded components, we add these components as children. Thereby, the tree contains on every even levels component nodes and on every odd levels fields.

The advantage of this representation is that it is simple to maintain the consistency of the field layer. For example, if the boundary between two fields is changed, we don't have to modify the contour in both polygons, but

rather only in the primal-dual graph. With the secondary data structure, it is also possible to check, that the new boundary does not break or change the embedding relations between the fields.

3.3.3 Recognizing dotted segments

The fields are usually colored with one color or has simple texture in topographic maps. For example, the sand and mud areas are marked with brown dots in a Russian map type. However, the dotted fields compose segments logically, the recognizable objects, i.e. the brown dots are not connected in the image, therefore the pixel-wise methods cannot be used in this particular case.

We used the following idea to recognize the dotted segments. First we recognize the dots in the image, and create a point set of the centers of the dots. Afterward, the point sets are clustered to form fields in the map. Generally, this algorithm can be used if the statements are satisfied:

- There is a recognizable property of the segment, which occurs densely in the segment.
- The property does not occur outside of the segment.

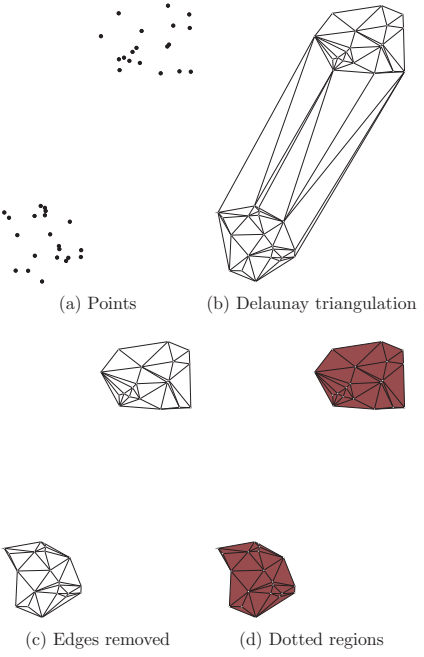
The first step of the algorithm is to identify the brown dots in the image. The brown color pixels can be determined with the method of 3.4.1 section. The brown connected components are considered as brown dots, if their pixel number are under a given threshold value. The brown dots are characterized by the center of the corresponding pixels.

The next step is to extract the segment polygons from the point set. A straightforward implementation would be to cluster the points into disjoint sets, and to use the convex hull for representing a segment.

This approach would result poor quality segmentation, because the concave segments would transformed to their convex hulls, and also the undotted internal holes in dotted segments would be classified as part of the segment.

Instead of the convex hull algorithm, we used Delaunay triangulation to construct the polygons of the dotted areas. The Delaunay triangulation tries to avoid the degenerate triangles, because the minimum angle of all triangles is maximized. More precisely, if a triangulation is described by the sequence of the angles in increasing order, then the triangulation is a Delaunay triangulation if and only if the sequence is lexicographically maximum.

The presented approach makes the clustering and the polygon extraction in one step. Let's take the Delaunay triangulation of the brown point set,



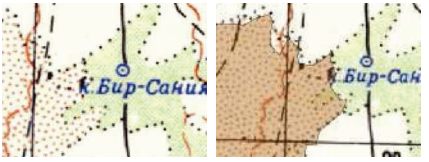
(a) Points (b) Delaunay triangulation

(c) Edges removed (d) Dotted regions

Figure 3.6: Dotted segment extraction with Delaunay triangulation

and remove those edges which are greater than a threshold value. The dotted region can be get by the union of all polygons smaller than a threshold.

The Delaunay triangulation can be computed in $O(n \log n)$, where n is the number of points. We used Fortune's algorithm [25] in our implementation. The result of a real example can be seen in the figure 3.7.



(a) Original map (b) Recognized dotted region

Figure 3.7: Dotted pattern recognition in Russian map

3.4 Layer of lines

The roads, railroads, power lines and some geological shapes are represented by lines in the map, in addition the level contours are also line objects.

3.4.1 Color decomposition classification

The topographic maps are printed with a limited number of different colors. The classification of the pixels to these color classes is a straightforward step during the preprocessing. This task seems to be similar to the classification of the satellite images, but there are important differences:

- The colors of a map type are fixed, therefore supervised segmentation algorithm can be used.
- The line and point objects are printed over the field objects, therefore the objects appear not in their distinct colors, but rather as mixtures of the field color and the object color.

These assumptions are formalized in the following image model. Let's define the set of field colors (C_f) and the set of object colors (C_o). Each pixel of the image can be get as a linear combination of a field and an object color.

To determine the class of a pixel, a least-square optimization is used:

$$\min_{c_f \in C_f, c_o \in C_o, \alpha \in [0,1]} \|(1-\alpha)c_f + \alpha c_o - c\|$$

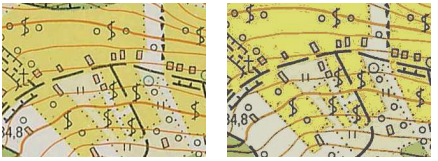


Figure 3.8: Color decomposition for line detection

Because both the number of field and object colors are small, therefore the optimum can be determined by solving the minimization problem for the each possible color pairs, and calculating the minimums of the quadratic functions of α.

The optimum solution is a field and an object color and an α value. If α is greater than α_0 threshold, then we consider that the pixel belongs to the object color, otherwise to the field color.

The output of this algorithm depends also on the used color-space. We tested RGB and HSL color models, and both of them gave satisfactory results. For the HSL model, the algorithm was slightly modified to consider the periodic character of the Hue component.

The result of the color decomposition and the whole preprocessing can be seen in Fig. 3.8 and 3.9.

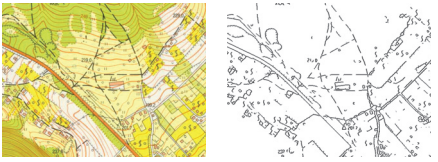


Figure 3.9: Preprocessed image for line recognition

The algorithm can be made more robust, if the neighbors of the pixels are also used.

3.4.2 Vectorization of curves

Before the vectorization, a thinning filter is applied, and the lines are converted to vector data by an edge detection algorithm. The received line strips

are simplified in order to describe the layer with respectable amount of line segments (see Fig. 3.10).

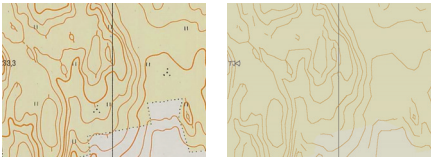


Figure 3.10: The vectorized line segments

The dashed line processing requires another algorithms. After the vectorization of the line segments, a filter is used to select those pieces which may be parts of dashed lines. The endpoints of the segments are connected by a weighted graph matching algorithm, where the edges are the potential connections, and the weights are calculated by the length of the segment and the deviation of the line directions.

3.5 Layer of symbols

In the symbol layer processing step the same color decomposition method is used as in the line layer processing (see Fig. 3.11), but of course, the symbol colors are used instead of the line colors. In order to recognize a symbol, a binary image pattern is created, on which the pixels with the color of symbol are marked.

3.5.1 Raster matching

The shape of the symbol is searched in the binarized image, and on each position the pixel-wise error is calculated. The costs of the inappropriate matching are chosen differently, if a symbol pixel is detected as a background pixel in the image, then the overall error is increased with $\frac{1}{2n_s}$, where n_s is the number of symbol pixels in the symbol shape. If the background pixel is treated as symbol pixel in the image, the error value is increased by $\frac{1}{2n_b}$, where n_b is the number of the background pixels in the symbol. By this cost function both of the empty and full regions get $\frac{1}{2}$ total error. Those positions are recognized as symbols where the error is less than a threshold, and there are no more symbols in a given distance.

The binary matching algorithm is made faster with a heuristic solution. Before the global matching test, preliminary tests are carried out. The number of symbol pixels and background pixels for the whole region and for each line are pre-calculated. Based on these values, a lower bound on the error is determined, so that an inadequate matching could be rejected in an early stage.

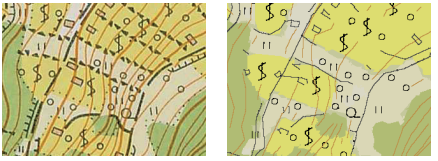


Figure 3.11: Object recognition

3.5.2 Recognizing rectangular shapes

General geometric matching, i.e. vector-based matching was tested, but the overall result was unreliable. However, for special purpose the geometric matching was satisfactory. In Russian maps, rectangular buildings have to be recognized, but neither the size nor the orientation of the buildings are known ahead.

The algorithm starts also from the binarized image, and we search the connected components of the symbol colored objects. If the object is a rectangular raster object, then its smallest covering rectangle is not much larger than the object size.

The rotating calipers algorithm is used to determine the minimum enclosing rectangle algorithm [68] of the objects. If the rate of the covering rectangle size and the object's pixel size is less than a threshold, and the aspect ratio of the covering rectangle is smaller than another threshold, then we accept the object as a building symbol.

The result of the building recognition can be see in Fig. 3.12.

3.6 Postprocessing

In the postprocessing step the recognized objects of different layers are used coherently to improve the overall quality of the map. Making corrections is part of this step. For example, when a symbol is recognized on a surface

Figure 3.12: Rectangular building detection

where it could not arise, e.g. a grape sign in a blue field, it will be removed from the map.

On other hand, some structural information can be retrieved. For example, the attributes of the field objects are extended with respect to the symbols thereon. If a grape sign is on a yellow field, the whole region is marked as a vineyard.

3.7 Conclusion

The main results of this section can be summarized as follows:

- A software framework was developed for creating automatically suitable vector maps from raster images, and it was customized to a workflow to convert Hungarian topographic maps to vector data.
- Although the representation of spatial objects with planar graphs is a known technique, it is rarely used GIS-applications. The most of the commercial software and file formats do not support this kind of data representation. The presented conversion framework show that this data structure helps to maintain the consistency of the map objects but it also keeps the ability to easily modify the map.
- Algorithm was developed for recognizing dotted segments of maps using Delaunay-triangulation.
- Algorithm was developed for recognizing rectangular shapes based on the minimum enclosing rectangle algorithm.

It should be also remarked, that with general algorithms a certain level of conversion quality can be reached, but for high quality results the human interaction is still necessary. With more sophisticated algorithms and custom handling of specific object types the automatic conversion can be improved, and can be significantly decrease the need of the human cost.

4 Optimization of work schedule of agents

In this section, we discuss a problem of optimizing the working schedule and the traveling route of agents.

There are given some business agents, who have to visit some customers at given times. Additionally, when the agents are not visiting customers, they have to do other jobs in their office. The task is to plan the work schedule of the agents for a single day. The presented solution is described in [17].

The workday regularly starts when the agent arrives at the office or at the place of the first meeting. Between the meetings or office shifts, they have to travel to the next location (or to the office) either by car or by public transport depending on the agents. The agents must also be allowed to have a lunch within a specified period.

On this problem, the author has worked together with Alpár Jüttner and Péter Kovács. The author's contribution was the model construction, and he has implemented several parts of the optimization software, e.g. the column generation method and the resource constrained shortest path algorithm.

4.1 Model of the scheduling problem

The master problem (MP) is formalized as an integer program of exponential size. Let A be the set of the agents and M be the set of meeting requests. Let S_a be the set of the feasible schedules of the agent $a \in A$, and c_{s_a} is the cost of a schedule $s_a \in S_a$. Then the integer programming formulation is the following.

$$\min \quad \sum_{a \in A} \sum_{s_a \in S_a} c_{s_a} x_{s_a} \tag{4.1a}$$

$$\text{s. t.} \quad \sum_{s_a \in S_a} x_{s_a} = 1 \qquad \forall a \in A \tag{4.1b}$$

$$\sum_{m \in s_a} x_{s_a} = 1 \qquad \forall m \in M \tag{4.1c}$$

$$x_{s_a} \in \{0, 1\} \qquad \forall a \in A, s_a \in S_a \tag{4.1d}$$

The cost c_{s_a} can be calculated from the properties of the schedule. If the agent travels in t_t minutes, waits for meetings in t_r minutes and works in t_w minutes in the office, moreover, there are n_c changes between meetings and office shifts, then we can compute the overall cost with the following expression:

$$c_{s_a} = C_t t_t + C_r t_r - B_w t_w + C_c n_c, \tag{4.2}$$

where C_t, C_r, B_w and C_c are the costs and the benefits of traveling, waiting, working in the office and changing between work types, respectively. These constants should represent the preferences of the customer.

We assign a binary variable x_{s_a} to each possible schedule s_a of each agent a. The constraints enforce that each meeting is fulfilled by exactly one agent and each agent has exactly one work schedule.

In addition, let *RMP* denote the following linear programming relaxation of the above formulation with the introduction of slack variables for constraints (4.1c) in order to ensure that the problem is always feasible.

$$\min \quad \sum_{a \in A} \sum_{s_a \in S_a} c_{s_a} x_{s_a} + \sum_{m \in M} H y_m \tag{4.3a}$$

$$\text{s. t.} \quad \sum_{s_a \in S_a} x_{s_a} = 1 \qquad \forall a \in A \tag{4.3b}$$

$$\sum_{m \in s_a} x_{s_a} + y_m = 1 \qquad \forall m \in M \tag{4.3c}$$

$$x_{s_a} \geq 0 \qquad \forall a \in A, s_a \in S_a \tag{4.3d}$$

$$y_m \geq 0 \qquad \forall m \in M \tag{4.3e}$$

where H is a sufficiently large number. The dual linear program of RMP is the following.

$$\max \quad \sum_{a \in A} w_a + \sum_{m \in M} z_m \tag{4.4a}$$

$$\text{s. t.} \quad w_a + \sum_{m \in s_a} z_m \leq c_{s_a} \qquad \forall a \in A, \forall s_a \in S_a \tag{4.4b}$$

$$y_m \leq H \qquad \forall m \in M \tag{4.4c}$$

$$w_a \in \mathbb{R} \qquad \forall a \in A \tag{4.4d}$$

$$z_m \in \mathbb{R} \qquad \forall m \in M \tag{4.4e}$$

The proposed solution is based on an iterative rounding of the LP relaxation of the above formulation. We fix the schedule of one agent in each iteration. For this, (a) we find an (approximate) solution to RMP, then based on this solution, we choose an agent and (b) fix a schedule for her by rounding the corresponding fractional variables $x_{s_a},\ s_a \in S_a$. Then we delete this agent and the requests covered by her schedule from the problem and repeat steps (a) and (b) until we find a schedule for each agent.

4.2 Solving RMP

To deal with the fact that the size of RMP can be enormous (exponential in the number of the meeting requests), we use the usual column generation approach [49], as briefly described below.

Instead of keeping the whole problem in the memory, we maintain a reasonably sized subset of the columns, i.e. a limited number of possible schedules $S'_a \subset S_a$ for each agent $a \in A$. Then we find a solution x_{s_a} to this subset of RMP together with the corresponding dual solution w_a, y_m. Now, we check whether (4.4b) holds for all agents $a \in A$ and for all schedules $s_a \in S_a$. If yes, then x_{s_a} is in fact the optimal solution. If not, then we add the column corresponding the failed dual constraint to the problem and iterate the above steps until the optimum is found or a certain time limit is reached.

Therefore, the core of this scheme is the subroutine that checks the feasibility of a dual solution and finds a failed constraint if it is not feasible.

The pseudo-code of the algorithm is the following.

```
lp ← create_lp(dir = MIN)
for m ∈ M do
  meeting_rows[m] ← lp.add_row(min = 1, max = 1)
  lp.add_col(meeting_rows[m], min = 0, max = 1, cost = H)
end for
for a ∈ A do
  agent_rows[a] ← lp.add_row(min = 1, max = 1)
  dual_expr, cost ← get_heuristic_schedule(lp, meeting_rows, a)
  lp.add_col(dual_expr + agent_rows[a], min = 0, max = 1, cost)
  dual_expr, cost ← get_empty_schedule(lp, meeting_rows, a)
  lp.add_col(dual_expr + agent_rows[a], min = 0, max = 1, cost)
end for
lp.solve()
was_change ← true
while was_change do
  was_change ← false
  for a ∈ A do
    column, cost, reduced_cost ← find_column(lp, meeting_rows, a)
    if reduced_cost − lp.dual(a) < 0 then
      lp.add_column(column + agent_rows[a], min = 0, max = 1, cost)
      lp.solve()
      was_change ← true
    end if
  end for
end while
```

First, the LP model consists of the constraints for the agents (4.3b) and the meetings (4.3c). Slack variables are added for the meeting rows and initial columns for the agents. For each agent, two initial columns are generated, one of them contains an empty schedule, i.e. the agent does not visit meetings during the day, which is used to maintain the feasibility of the LP model in the rounding phase (see Section 4.3). The second column is created with a greedy heuristic, decreasing the steps of the column generation algorithm significantly.

Afterward, we look for columns with negative reduced cost, i.e. columns which violate the dual constraint (4.4b), and we add them to the LP model. If there is no violating column in one whole iteration, then the solution of the RMP is optimal.

4.2.1 Column Generation with Dynamic Programming Method

During the algorithm, we have to find feasible schedules which violate the inequality (4.4b). It can be done by searching the schedule with the smallest reduced cost for agent a, i.e.

$$\min c_{s_a} - w_a + \sum_{m \in s_a} z_m \tag{4.5}$$

The problem can be formulated as finding a resource constrained shortest path. Let us define a directed graph $G = (V, E)$ as follows. We assign a node in the graph to each meeting request. Between two meetings m_1 and m_2, the agents can do only a couple of different things. We add a directed edge $m_1 \longrightarrow m_2$ to each of these actions whenever it is feasible (parallel edges are allowed), and assign the cost corresponding to the action. Namely, the choices are the following.

- The agent can go directly to the place of m_2.
- If there is enough time between the meetings, she can go to the office for internal work.
- The agent can also have a lunch time between the meetings.

So, the traveling time is calculated between every pair of meetings m_1 and m_2, both directly and through the office location. The travel time has to fit in to the time gap, and the travel lengths must be shorter than a threshold value. Moreover, if the agent goes to the office, she has to stay there for at least a certain time period, which is necessary to do effective work.

Each path in this graph almost determines a schedule, but the periods before the first and after the last requests are not fixed. Therefore additional edges are assigned to the graph which belong to the exterior schedule. The schedules must consist of an internal path and from an external edge between the same starting and finishing nodes.

A cost function $c : E \to \mathbb{R}$ is defined on the edges, which can be calculated based on equation (4.2). The cost of a schedule can be determined as the total cost on the corresponding path and external edge. To get the reduced cost of a schedule, we have to subtract the dual values z_m from the path cost.

The agents' working time must include the lunch time. Thus we assign another function $r : E \to \{0, 1\}$ to the edges depending on whether it includes a lunch period (the external edge may include a lunch period, as well).

Claim 3 *The optimal schedule is the minimum cost feasible combination of the internal paths and external edges for all pairs of nodes.*

We use label setting resource constrained shortest path algorithm [35] for finding a schedule with minimum cost and exactly one lunch period. Since both an internal path and an external edge have to be find, we calculate the shortest paths on the internal edges between each pair of nodes in the graph. Note that the edges are directed from an earlier event to a later one, thus the walks obtained by the label setting algorithm are simple paths.

The label setting method is not polynomial in general, but in our case, the resource function has only two values, which ensures the polynomial time complexity.

Claim 4 *The dynamic programming algorithm for column generation runs in polynomial time.*

If n is the number of the meeting requests, then the graph can have $O(n^2)$ edges. Therefore the shortest paths from an arbitrary starting node can be calculated in $O(n^2)$ time. This yields $O(n^3)$ total running time for finding the optimal schedule.

During the real life application of the presented method, various new requirements was requested by the customer.

A general goal is that a balanced schedule has to be achieved, i.e. both office and visit shifts should be assigned to each agent. Additionally, the experienced agents should get more meetings in a day, while at most one visit should be assigned to the beginners. These requirements are assumed to be soft constraints, thus they are encoded into the objective function.

Such constraints can be handled easily with introducing new resources in the dynamic programming algorithm. The number of meetings and the

existence of an office shift are registered in the labels. If the meeting number exceeds or does not reach the limit, then the cost of the schedule is increased proportionally to the deficit or the surplus. Moreover, if the agent does not have office or meeting shift, an additional constant is added to the schedule cost value. Since the number of meetings is limited by a small constant, which is independent of the number of agents and requests, the modified algorithm is also polynomial.

4.2.2 Point-to-point shortest path

The algorithm calculates traveling times between the meetings, which yields an additional optimization problem. In the presented solution, third party service was used to get a route for the agents, but in this section, the basic algorithmic background is provided for the point to point path search problem.

The map is described as a graph, where the nodes are the crossroads and the edges are the road sections. The minimum travel time problem can be formulated as a shortest path problem, if the travel time of the road sections are used as the weights of the edges.

The shortest path search is a well-studied graph optimization problem. We present here the Dijkstra algorithm and some heuristic improvements.

The Dijkstra algorithm [13] maintains an estimated distance function $d_s : V \to \mathbb{R}_0^+$ from the starting node and a predecessor function $p_s : V \to V$. The pseudo-code of the algorithm is the following.

for $v \in V$ **do**
  $d_s(v), p_s(v) \leftarrow \infty,$ **undefined**
end for
$d_s(s) \leftarrow 0$
$Q \leftarrow V$
while Q **is not empty do**
  $u \leftarrow \min_{u \in Q} d_s(u)$
  $Q \leftarrow Q \setminus u$
  if $u = t$ **then**
    return $d_s(t)$
  end if
  for $a(u, v) \in \mathrm{A}_{\mathrm{out}}(u)$ **do**
    if $d_s(u) + w_a < d_s(v)$ **then**
      $d_s(v), p_s(v) \leftarrow d_s(u) + w_a, u$
    end if
  end for
end while

The Dijsktra algorithm builds a "ball" of visited vertexes around s with $d(s,t)$ radius. The number of discovered nodes can be decreased significantly with the $A^\star$ algorithm [57]. We need a $\pi_t : V \to \mathbb{R}$ potential function, which satisfies the $w_{a(u,v)} > \pi_t(u) - \pi_t(v)$ inequality.

for $v \in V$ **do**
  $d_s(v), p_s(v) \leftarrow \infty$, **undefined**
end for
$d_s(s) \leftarrow 0$
$Q \leftarrow V$
while Q **is not empty do**
  $u \leftarrow \min_{u \in Q} d_s(u) + \pi_t(u)$
  $Q \leftarrow Q \setminus u$
  if $u = t$ **then**
    return $d_s(t)$
  end if
  for $a(u,v) \in \mathrm{A}_{\mathrm{out}}(u)$ **do**
    if $d_s(u) - \pi_t(u) + w_a < d_s(v) - \pi_t(v)$ **then**
      $d_s(v), p_s(v) \leftarrow d_s(u) + w_a, u$
    end if
  end for
end while

It can be seen, that this algorithm is equivalent to the shortest path search in the same graph but with $w'_a = w_{a(u,v)} - \pi_t(u) + \pi_t(v)$ weight function. If the potential function approximates well the real distance of node t, then the algorithm visits significantly less nodes than the Dijsktra algorithm and provides better running time. In the traveling time case, a simple potential function can be calculated from the physical Euclidean distance and the maximum speed limit.

Alternative potential function is suggested by Goldberg and Harrelson [33]. They choose K nodes from the graph and compute the shortest paths for each node of the graph from and to the selected nodes. With using the triangle inequality, we can obtain the lower bound on the distance $d(u,v) \geq \min_{l \in L}(d(u,l) - d(v,l))$, where L is the set of the selected nodes. The selected nodes can be chosen either randomly in the graph, or greedy method is suggested in the original article.

The Dijkstra algorithm can also be improved if the search applied both from the source and to the target parallel [20]. The algorithm extends the visited nodes from both starting points alternately (or by minimum distance), and when a node gets its final distance from both direction, the algorithm can finish the execution. The optimal path does not go through the common

node necessarily, but it can also connect a node with fixed distance from the source and another node with fixed distance to the target.

The idea of the bidirectional Dijkstra algorithm can also be used in the $A^\star$ algorithm [33]. In this case, two potential function should be provided, one which estimates the distance to the target node ($\pi_t(u)$), and the other one estimates the node distances from the source node ($\pi_s(u)$). In addition, these potential function should satisfy the consistency criteria, i.e. the $\pi_s(u)+\pi_t(u)$ equals to a constant for each u node.

4.3 Rounding Phase

Once we have a solution for RMP, the rounding phase is easy. We simply evaluate each of the K largest primal variables, and compute the increase in the cost function when this variable is set to 1 and all others for the corresponding agent are set to 0. We choose the one resulting in the least increment and fix the corresponding schedule s_a for the agent a. Then we remove agent a and all covered requests from the problem and repeat the column generation and rounding phases until the schedule of each agent is fixed.

The schematic code of the algorithm is as follows.

```
fixed_cols ← set()
for k ∈ {1 ... |A|} do
  candidates ← array()
  for c ∈ lp.cols() do
    if c ∉ fixed_cols then
      candidates.add(pair(lp.primal(c), c))
    end if
  end for
  candidates ← n_largest(candidates)
  fixing_col, min_cost ← null, +∞
  for (_, c) ∈ candidates do
    lp.set_min(c, 1)
    lp.solve()
    if lp.objective_value() < min_cost then
      fixing_col, min_cost ← c, lp.objective_value()
    end if
    lp.set_min(c, 0)
  end for
  lp.set_min(fixing_col, 1)
  lp.solve()
```

```
    fixed_cols.add(fixing_col)
    reoptimize_with_colgen(lp)
  end for
```

On the implementation level, the fixed variables and the already satisfied constraints are not removed from the LP model but the lower bound of the fixed variable is adjusted to value 1. For revised primal simplex method, this ensures that the previous basis is still usable for the re-optimization. Because there is slack column for each meeting and empty schedule for each agent, the LP problem remains feasible after each fixation.

The function `reoptimize_with_colgen`() uses the same method as the algorithm in subsection 4.2. Similar greedy rounding technique is used in airline crew scheduling [5], as well.

4.4 Implementation Details

This optimization problem came from the real life and the proposed solution is currently in production use. To develop a reliable and efficient solution, a couple of technical issues must be dealt with.

Because of running time efficiency, the algorithm was implemented in C++ language, and several third party libraries were used. The solver-independent LEMON LP and MIP interface, which is described in details in Section 5.1.7, turned out to be very useful. As the backend of the LP interface, the GLPK library [32] was used.

Estimating the travel times between the geographical locations is a major difficulty in practice, since it needs large scale up-to-date road network data. Accurate values can be obtained using on-line route planner services, but it is a costly operation, both in terms of money and in terms of time. Therefore a mixed strategy was developed for this purpose. First, a simple straight line distance-based estimation is used in the dynamic programming algorithm (using a pessimistic calibration), but the accurate travel times are queried when a column is added to the problem. Moreover, once a travel time between two locations has been queried, this value is stored and it will be used even in the dynamic programming method later on.

4.5 Experimental Results

The algorithm has been evaluated with respect to the running time and objective value aspects using 10 real-life input instances. In these test cases, the number of the agents is between 75 and and 95, and the number of meeting requests is between 100 and 185.

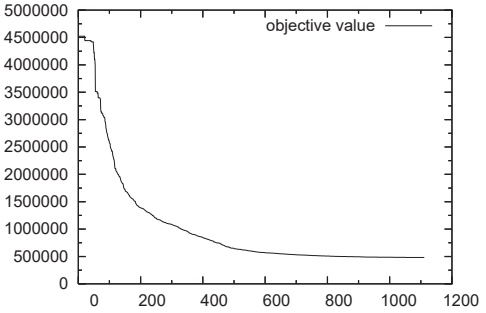


Figure 4.1: Objective value in the improvement phase

Fig. 4.1 shows the change of the objective value in the improvement phase. For numerical comparison, the optimal solution of the relaxed problem was also calculated. The experiments show that the gap between this value and the obtained integer solution is quite small. It also means that the optimal fractional and integer solutions are usually close to each other for this problem.

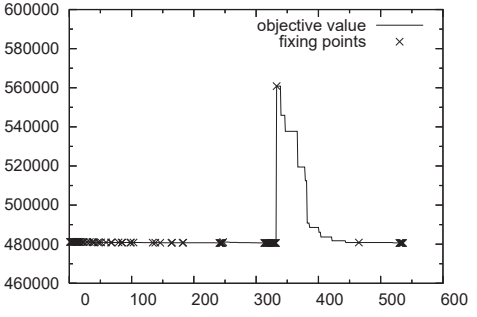


Figure 4.2: Objective value in the rounding phase

Fig. 4.2 shows a typical result of the rounding phase. Most of the fixations

do not raise the objective value, therefore they follow each other directly. On the other hand, some fixations increase the objective value significantly, but it can be decreased back close to its original value by some improvement steps.

The algorithm was tuned by setting the maximum number of generated new columns per rounding phase in order to meet the customer's request. For usual input sizes, the results are obtained within a couple of minutes. This running time is dominated by the web service access.

4.6 Conclusion

The main results of this section can be summarized as follows:

- A complex optimization method for a work schedule planning problem has been developed.
- The heuristic rounding method for the column generation algorithm was evaluated.
- The experimental results show that the algorithm is a suitable and efficient solution for the agent scheduling problem.

5 LEMON optimization library

The LEMON library stands for **L**ibrary for **E**fficient **M**odeling and **O**ptimization in **N**etworks. It is a highly efficient open source graph template library that provides implementations of data structures and algorithms related to graph optimization [47, 18].

In geoinformatics and remote sensing, the real-world objects can be naturally represented by graphs, that is why LEMON was used in several research projects, for example, in the graph-based segmentation algorithms, for representing vectorized maps and for solving linear programming problems and resource constrained shortest path tasks in the agent scheduling problem.

The LEMON project was initiated by the Egerváry Research Group on Combinatorial Optimization (EGRES) [23] at the Department of Operations Research, Eötvös Loránd University, Budapest, Hungary in 2003. It became member of the COIN-OR initiative [11] in 2009. The project is led by Alpár Jüttner, and it has had several contributors. Particularly, Péter Kovács has had a big impact on quality of the stable release branch.

The author joined to the LEMON Project in 2004, and he has made many contributions to the project:

- participation in the design and implementation of graph structures
- implementation of various graph and map adaptors
- design and implementation of path data structures
- implementation of bucket and radix heap
- implementation of heap union-find data structure
- implementation the Bellman-Ford algorithm
- rewriting the preflow algorithm
- implementation of minimum cost arborescence algorithm
- implementation of algorithms for minimum cut problems (Hao-Orlin, Nagamochi-Ibaraki and Gomory-Hu)
- implementation of algorithms for matching problems in general graphs (maximum cardinality/weighted/perfect weighted, fractional/integral)
- implementation of algorithms for checking graph properties (connectivity, strong connectivity, bi-connectivity, bipartiteness)

- implementation of linear time planar graph embedding and drawing algorithms
- implementation of radix sort
- design for adding MIP solver interface to LEMON
- adding solver interface for SoPlex, CLP and CBC
- design and implementation of the reader and writer for LEMON graph format file

Note that, in the LEMON stable branch, every code has been peer reviewed and the design has been discussed between the team members. Therefore, each part of LEMON must be considered as result of team work.

5.1 The structure of LEMON library

LEMON implements several data structures, which can be used in optimization algorithms. The most important types are the graph classes and the corresponding data types, like nodes, arcs and property maps. In addition, LEMON also provides several auxiliary data structures, like heaps, union-find structures.

Following the generic programming paradigm, LEMON defines the algorithms separately from the data structures. Numerous efficient algorithms are implemented, such as graph search and shortest path algorithms, flow computations and matching, etc.

5.1.1 Graph Data Structures

Although LEMON is a generic library, its main graph types are not template classes, which is made possible by an important design decision. Namely, all data assigned to nodes and arcs are stored separately from the graph data structures (see Section 5.1.3).

The most important graph type is the `ListDigraph`, which is a general directed graph implementation based on doubly-linked adjacency lists. Another important digraph-type is `SmartDigraph`, which stores the nodes and arcs continuously in vectors and uses simply-linked lists for keeping track of the incident arcs of each node (see Section 5.2.1). Therefore, it has smaller memory footprint than `ListDigraph` and can be considerably faster, at the cost that nodes and arcs cannot be removed from it. `ListGraph` and `SmartGraph` are the undirected versions of these data structures.

LEMON follows the generic programming paradigm similarly to Boost Graph Library (BGL) [4, 30] and LEDA [75, 51], and describes the requirements of generic components by means of *concepts*. Concepts play the same role as in STL; they define the supported functionality of data types, along with their user interfaces and semantics.

LEMON defines two graph concepts, `Digraph` and `Graph`, which describe the requirements for directed and undirected graphs, respectively. The undirected `Graph` concept is designed to also satisfy the requirements of the `Digraph` concept in such a way that each *edge* of an undirected graph can also be viewed as a pair of oppositely directed *arcs*. Therefore, each undirected graph – without any transformation – can be considered as a directed graph at once.

The main benefit of this design is that all directed graph algorithms automatically work for undirected graphs, as well. In most cases, this also means that there is no need for separate algorithm implementations. However, particular algorithms could require specialization for undirected graphs. Such a special method is the checking of the Eulerian property, which is discussed in detail in Section 5.2.4.

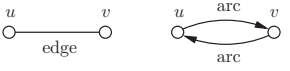


Figure 5.1: Undirected edge as two directed arcs

Undirected graphs provide an `Edge` type for the undirected edges and an `Arc` type for the arcs. This separation makes the implementation of some algorithms simpler (e.g., planar graph algorithms) because we can distinguish the undirected edges from their directed variants. On the other hand, the `Arc` type of an undirected graph is convertible to the `Edge` type, thus the corresponding edge of an arc can always be obtained conveniently, without calling any functions. As a result, all methods and data structures that are designed for edges can be used directly with both edges and arcs. This can be practical in several cases. For example, a property map (see Section 5.1.3) that assigns data to edges can be used with both edges and arcs (but an arc map can only be used with arcs).

Comparison to BGL implements a single adjacency list-based graph class which can be fully customized with template parameters that specify the internal storage data structures for nodes and arcs. Furthermore, a graph can be set to be *directed*, *bidirectional* or *undirected* using another template parameter. The bidirectional graph concept is the equivalent of LEMON's `Digraph` concept. These graph types support traversing through the outgoing

and incoming arcs of each node. The directed graphs of BGL store only the outgoing arc lists, thus they require less storage space than bidirectional graphs. Note that this category is missing from LEMON.

The `adjacency_list` class template of BGL implements both directed and undirected graphs by extensive use of template specializations. As a result, directed and undirected graphs have the same interfaces but different semantics in BGL. The edges of undirected graphs are usually considered undirected, but they have directions in some cases, for example, in iterations. Such an inconsistency is often confusing. Moreover, this design does not make it possible to define property maps whose keys are the directed variants of the edges, although it would also be important in certain algorithms.

LEDA's general `graph` class has closed source, but its implementation is probably similar to the general graph types of LEMON. The main difference is that LEDA implements directed and undirected graphs in the same class and provides member functions to switch between the two modes. This design is certainly convenient in some cases, but it is less distinctive than LEMON's concepts, therefore, it has some disadvantages, similarly to BGL.

LEMON also provides a few special graph classes, like full or grid graphs. These graph types do not store the adjacency lists but only the parameters of the graph, i.e. the number of the nodes for full graph and the width and height for grid graphs. Based on these parameters the classes provide all operations on-the-fly. The grid graph types can be used to naturally represent images (see also Section 2.4).

Furthermore, each of the three libraries provides an optimized static data structures for directed graphs, which stores the nodes and arcs in arrays or vectors in such a way that the arcs are sorted by their source nodes. As the crucial operations of most directed graph algorithms iterate on the outgoing arcs of the nodes, they typically run faster using these static implementations.

5.1.2 Iterators

Most graph libraries provide iterator classes for traversing through the elements of the graph data structures (i.e., the nodes and arcs). LEMON defines a special iterator interface, which does not conform to the iterator concepts of the C++ Standard Template Library (STL).

The iterators of LEMON are initialized to the first element in the traversed range by their constructors, and their validity is checked by comparing them to a special constant `INVALID`. Furthermore, each iterator class is convertible to the corresponding graph element type, without having to use `operator*()`. This feature distinguishes LEMON iterators from the standard C++ iterators and makes their usage slightly simpler.

For example, the distance value of each node can be printed to the standard output as follows.

```
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) {
  std::cout << g.id(v) << ": " << dist[v] << std::endl;
}
```

In the first line, all occurrences of `v` refer to the iterator itself, while the corresponding node object is referred twice inside the loop.

Note that this concept could not be applied to general iterators. For example, STL defines iterators for containers of arbitrary items. It means that the iterator type and the item type of the container could have conflicting functionality, for instance, both of them could support `operator++()`. Therefore, an iterator object and the referred object must be distinguished: `it++` affects the iterator `it`, while `(*it)++` affects the referred object `*it`.

LEMON iterator concept, however, exploits the speciality of graphs, which can be viewed as containers of particular elements. The nodes and arcs themselves provide a strongly limited set of features, which does not conflict with the functionality of iterators. Therefore, the program context always indicates whether we refer to an iterator or to a graph element, as we have already seen in the above example.

In contrast with this, BGL iterators follow the STL requirements of input iterators. It means that they must be dereferenced with the `operator*()` function to obtain the corresponding item descriptors.

After running Dijkstra's algorithm, the node distances can be printed as follows.

```
graph_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi) {
  std::cout << *vi << ": " << dist[*vi] << std::endl;
}
```

The `tie()` function is used to make the code more compact and to avoid simple mistakes of the programmer.

A drawback of the above solution is that the iterator objects are defined in a wider scope than the loop itself. BGL, however, also provides several iteration macros that simplify traversing graph elements and define the loop variables only in the scope of the loop.

```
vector<int> dist(num_vertices(g));
BGL_FORALL_VERTICES(v, g, graph_t) {
  std::cout << v << ": " << dist[v] << std::endl;
}
```

Similar macros are available in LEDA, but they do not allow to define the loop variables only in the scope of the loop.

```
node v;

forall_nodes(v, g) {
  g.printNode(v);
  std::cout << ": " << dist[v] << std::endl;
}
```

5.1.3 Handling Graph Related Data

In addition to the pure graph data structures, most graph algorithms need additional data associated to the nodes and arcs. For example, shortest path algorithms require a length function on the arcs and record the computed distance labels for the nodes. Graph libraries support handling these associated values in various ways. The data structures used for this purpose are typically called *maps* (not to be confused with `std::map`, which provides a rather slow $O(\log n)$ time access to the elements). Since they are among the most frequently used data structures, maps should be highly efficient and convenient.

The most important operation of a map data structure is the access of its elements, that is, retrieving or overwriting the value assigned to a certain node or arc. In most graph libraries, time complexity of these operations is $O(1)$. Library designers have to deal with two additional performance considerations. First, map access operations should not be virtual functions because that forbids inlining. Second, it is worthwhile to use continuous storage for maps since it usually induces faster data access due to better caching.

LEMON features only external property maps that are stored separately from the related graph data structure, but they are updated automatically on the changes of the graph (see Section 5.2.3). The main advantage of external maps is their great flexibility. They can be constructed and destructed freely, so their lifespans are not bound to the lifespan of the graph. Moreover, separate storage could result in better caching properties, especially using several maps for a large graph.

Using LEMON, node and arc maps can be declared as follows.

```
ListDigraph::NodeMap<std::string> label(g);
ListDigraph::ArcMap<int> length(g);
```

The map values can be obtained and modified using the corresponding overloaded versions of `operator[]()`.

```
label[v] = "source";
length[e] = 2 * length[f];
```

In addition to the standard graph maps, LEMON also contains several "lightweight" *map adaptor* classes. They are not stand-alone maps with own data storage, but instead they adapt one or more other map objects and alter their data "on the fly". When the access operation of a map adaptor is called, it reads the corresponding data from the underlying maps and performs a certain operation on them, but without actually modifying or copying the original storage. These adaptor classes also conform to the map concepts, thus they can be used like standard LEMON maps.

Let us suppose that we have a traffic network stored in a LEMON graph object with two arc maps `length` and `speed`, which store the physical length of the corresponding road section for each arc and the maximum (or average) speed that can be achieved on it, respectively. If we are interested in the optimal traveling times, then we can call Dijkstra's algorithm as follows (In Section 5.1.4, the interface of the algorithm is discussed in details).

```
dijkstra(g, divMap(length, speed)).distMap(dist).run(s);
```

The `divMap()` function gives back a map adaptor object that provides the quotient of the values of the two original maps. It means that the Dijkstra algorithm receives the expressed traveling time of the corresponding road section for each arc.

Contrary to LEMON, several libraries store the associated data directly in the node and arc objects of the graphs. For example, only a limited number of internal maps of fixed types can be used in the Stanford GraphBase library [40, 59]. This design allows easier implementation of the core parts but makes harder developing complex algorithms and strongly limits the versatility of the library.

BGL supports both internal and external storage of graph related data. The interior properties of nodes and edges can be specified as *bundled properties* or *property lists*. The bundled properties provide a much simpler interface and their use is to be preferred, whereas the latter solution is compatible with older compilers and older versions of the Boost library.

If more than one assigned values are required for the nodes and edges, they have to be collected into specific data types, which are then passed as template parameters to the graph class.

```
struct NodeData { ... };
struct EdgeData { ... };

typedef adjacency_list<listS, vecS, bidirectionalS,
  NodeData, EdgeData> GraphType;
```

The main advantage of internal storage is that its capacity is adjusted automatically if the graph is modified, but it is not flexible as its lifespan is strictly bound to the graph object.

External property maps are also supported in BGL by wrapping standard container data structures. They are more flexible than interior properties since their lifespans are not bound to the associated graph. However, we have to choose between efficiency and convenience if we use these maps in conjunction with a varying graph. We can apply a map that wraps a random access container (e.g., `std::vector`) to ensure rapid data access, but it must be updated manually each time the graph changes. Alternatively, we can also use an external map that is based on an associative container (e.g., `std::map`). This solution naturally adapts to any change of the graph without explicit updating, but at a significant expense of efficiency. Note that LEMON's graph maps, however, provide this flexibility and convenience without the expense of performance (see Section 5.2.3).

LEDA implements two kinds of external data structures for handling graph related data. The *arrays* are static data structures, but their access operations take constant time. The *map* types are more flexible as they are not invalidated when the associated graph is changed. However, they are implemented by hash tables, and so they are less efficient. Therefore, we encounter the similar trade-off as with the external maps of BGL.

Although these data structures are external, LEDA makes it possible to allocate additional storage space for them in the graph objects. The newly created arrays and maps can be assigned to these slots, so the memory usage can be optimized. Apart from these solutions, LEDA also provides parametrized graph data structures, whose node and edge objects can contain arbitrary additional data, just like the bundled properties in BGL.

5.1.4 Algorithms

Inevitably, the most important differentiating factor between graph libraries is the range and quality of the implemented algorithms. A simple graph data structure for a specific use can be implemented rapidly, but sophisticated algorithms need careful design and lots of work from skilled programmers, especially when the efficiency is of high priority.

LEMON provides highly efficient implementations of numerous algorithms related to graph theory and combinatorial optimization. These algorithms include fundamental methods, such as breadth-first search (BFS), depth-first search (DFS), Dijkstra algorithm, Bellman-Ford algorithm, Kruskal algorithm, and methods for discovering various graph properties (connectivity, bipartiteness, Eulerian property, etc.), as well as complex

algorithms for finding maximum flows, minimum cuts, feasible circulations, maximum matchings, minimum mean cycles, minimum cost flows, and planar embedding of a graph. BGL and LEDA feature similar varieties of algorithms but with different interfaces.

In LEMON, algorithms are implemented as class templates, but for the sake of convenience, function-type interfaces are also available for some of them. For instance, Dijkstra's algorithm is implemented in the `Dijkstra` class, but a `dijkstra()` function is also defined.

```
dijkstra(g, length).distMap(dist).run(s);
```

The function interfaces of the algorithms are considerably simpler, but they are suitable for most practical cases due to the extensively used *named parameter* technique. This technique supports several function parameters with default values and an arbitrary set of these parameters can be specified in an arbitrary order by calling a dedicated function for each desired parameter. It means that the parameters are referred by names instead of the standard position-based reference. LEMON implements named parameters quite similarly to the Boost library [30].

The Dijkstra algorithm with class interface can be used as follows.

```
Dijkstra<ListDigraph> alg(g, length);
alg.distMap(dist);
alg.run(s);
```

This code is longer than the one using the function interface, but the execution can be controlled to a higher extent using this interface. For example, more source nodes can be specified and the algorithm can also be executed step-by-step, as the following code demonstrates.

```
alg.init();
alg.addSource(s);
while (!alg.emptyQueue()) {
  ListDigraph::Node v = alg.processNextNode();
  std::cout << g.id(v) << ": " << alg.dist(v) << std::endl;
}
```

The basic functionality of the algorithms can be greatly extended using special purpose map types for their internal data structures. For example, the `Dijkstra` class stores a `ProcessedMap`, which should be a writable node map of `bool` value type. The assigned value of a node is set to `true` when the node is processed, that is, its actual distance is found. Applying a special map, `LoggerBoolMap`, the processing order of the nodes can be recorded easily in a standard container.

Such specific map types can be passed to the algorithms using the technique of *named template parameters*. Similarly to the named function parameters, they allow specifying any subset of the parameters in arbitrary order.

```
typedef vector<ListDigraph::Node> Container;
typedef back_insert_iterator<Container> InsIterator;
typedef LoggerBoolMap<InsIterator> MyProcessedMap;

Container container;
InsIterator iterator(container);
MyProcessedMap map(iterator);
Dijkstra<ListDigraph>
  ::SetProcessedMap<MyProcessedMap>
  ::Create alg(g, length);

alg.processedMap(map);
alg.run(s);
```

Surprisingly, even the above example can be implemented using the `dijkstra()` function and named parameters as follows.

```
vector<ListDigraph::Node> container;
dijkstra(g, length)
  .processedMap(loggerBoolMap(back_inserter(container)))
  .run(s);
```

Note that a function interface has the major advantage that temporary objects can be passed as reference parameters. In this example, both the insert iterator object and the map object are created only temporarily, and the compiler will probably completely optimize them out from the final executable.

BGL implements several algorithms with *visitor-based* interfaces instead of using special purpose graph maps. The visitor classes are the generalizations of function objects: they have more entry points by defining several callback functions. A visitor-based algorithm emits different events during its execution and calls the corresponding entry functions of the associated visitor. In some cases, this technique could be more convenient than the use of customized maps, because all event handler operations are implemented in the same class. For this reason, LEMON also provides visitor-based solutions but only for the basic graph search algorithms, BFS and DFS.

LEDA provides less flexibility in using algorithms than the other two libraries. It implements a few compact function interfaces for each algorithm but without named parameters. These functions are designed for the most typical use cases and support only a limited set of configuration options.

5.1.5 Graph Adaptors

In typical graph algorithms and applications, we usually require a specific alteration of a graph. For example, certain nodes or arcs should be removed or the reverse oriented graph should be used. However, the actual modification of the physical storage or making a copy of the data structure along with the required maps could be rather expensive (in time or in memory usage) compared to the operations that should be performed on the altered graph. In such cases, LEMON's graph adaptor classes can be used.

Graph adaptors are special class templates that serve for considering other graph data structures in different ways. They are based on the same idea as the previously discussed map adaptors (see Section 5.1.3), but they are more complex. Graph adaptors can only be used in conjunction with another graph object that provides an actual storage of a graph. They do not modify the underlying data structure, they just give another view of it by utilizing the original operations. Graph adaptors conform to the graph concepts, thus they can be used the same as "real" graphs, and all generic algorithms works for them.

The following example shows how the `ReverseDigraph` adaptor can be used to run Dijkstra's algorithm on the reverse oriented graph.

```
dijkstra(reverseDigraph(g), length)
  .distMap(dist).run(s);
```

Note that the maps of the original graph (`length` and `dist`) can also be used with the adaptor, since the node and arc types of all adaptors convert to the original item types.

As this example suggests, graph adaptors help writing compact and elegant code and make it easier to implement complex algorithms based on reliable standard components.

Another fundamental graph alteration is the hiding of nodes and arcs, which can be achieved using one of the subgraph adaptors in LEMON. These classes store filter maps that are used by the iterators to skip the currently hidden items. Therefore, subgraph adaptors are significantly less efficient than the original graph objects.

As the adaptor classes conform to the graph concepts, we can even apply an adaptor to another one. Figure 5.2 illustrates a situation when a `SubDigraph` adaptor is applied to a directed graph and `Undirector` is used to make the obtained subgraph undirected.

Combinatorial optimization methods are usually based on more complex graph alterations. For example, the residual network is a particularly important model for flow and matching algorithms. `ResidualDigraph` implements

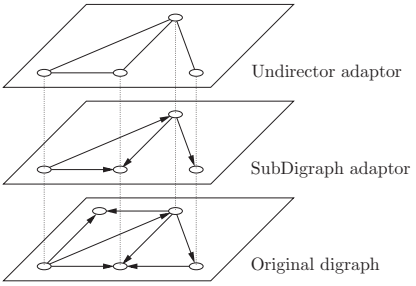


Figure 5.2: Illustration of graph adaptors in LEMON

this network by adapting a directed graph along with a capacity map and a flow map.

`SplitNodes` is another practical adaptor that splits each node into an in-node and an out-node in a directed graph. Formally, the adaptor replaces each node v with two nodes v_{in} and v_{out}. Each arc (u, v) of the original graph will correspond to an arc (u_{out}, v_{in}). The adaptor also adds an additional bind arc (v_{in}, v_{out}) for each node v of the original graph. The aim of this construction is to assign costs or capacities to the nodes of the graph when using algorithms which would otherwise consider only arc costs or capacities.

BGL also features graph adaptors but only a few basic ones, like `reverse_graph` or `filtered_graph`. On the other hand, LEDA does not provide similar features.

5.1.6 Auxiliary Tools

Graph algorithms often depend on various auxiliary data structures and algorithms. For example, priority queues play an important role in Dijkstra and Prim algorithms: both their theoretical and practical performance are determined by the applied data structures. Therefore, LEMON provides a wide variety of heap implementations for this purpose. In most algorithms, the underlying data structures are easily replaceable using the named template parameters. The following code demonstrates the usage of Dijkstra's algorithm with a Fibonacci heap.

```
typedef FibHeap<int, Digraph::NodeMap<int> > FHeap;
Dijkstra<Digraph>
  ::SetStandardHeap<FHeap>
  ::Create alg(g, length);
```

Several data structures are available for maintaining *disjoint sets*. `UnionFind` is the classical union-find data structure, which is used to implement the Kruskal algorithm. The `UnionFindEnum` and `HeapUnionFind` classes are used in matching algorithms, the first one supports the enumeration of the items stored in the sets, while the second one also assigns priorities to the elements and an item having minimum priority can be retrieved set-wise.

The `Elevator` classes are general purpose data structures to support the push-relabel-based algorithms, they can maintain the level and activeness of the observed items. The global relabeling, the highest label and the gap heuristics can be implemented easily with these tools.

5.1.7 LP Interface

Linear programming (LP) is probably the most fundamental general methods of operations research. Countless optimization problems can be formulated and solved using LP techniques. Nowadays, various efficient LP solvers are available, including both open source and commercial software. Therefore, LEMON does not implement its own solver but features wrapper classes for several LP libraries providing a common high-level interface for them.

The advantage of this design is twofold. First, LEMON applies an object oriented approach, which is quite similar to the ILOG Concert Technology [14]. This approach makes LEMON's interface more flexible than the native interfaces of several LP libraries and it could be more comfortable for those who are familiar with object oriented programming. Second, changing the underlying solver in an application that uses this common syntax needs virtually no effort. Therefore, one can easily experiment various LP solvers in her particular application and compare their efficiency at any stage of the development.

Let's consider the following linear programming task:

$$\begin{aligned} \max \quad & 10x_1 + 6x_2 \\ \text{s. t.} \quad & 0 \leq x_1 + x_2 \leq 100 \\ & 2x_1 \leq x_2 + 32 \\ & x_1 \geq 0 \\ & x_2 \leq 10 \end{aligned}$$

In LEMON, `Lp::Col` represents the variables of the LP problems, while `Lp::Row` represents the constraints. The numerical operators are used to form expressions from columns and dual expressions from rows. Due to the suitable operator overloads, an LP problem can be described conveniently, directly as it is expressed in mathematics:

```
Lp lp;
Lp::Col x1 = lp.addCol();
Lp::Col x2 = lp.addCol();

lp.max();
lp.obj(10 * x1 + 6 * x2);

lp.addRow(0 <= x1 + x2 <= 100);
lp.addRow(2 * x1 <= x2 + 32);

lp.colLowerBound(x1, 0);
lp.colUpperBound(x2, 10);

lp.solve();
std::cout << "Solution: " << lp.primal() << std::endl;
std::cout << "x1 = " << lp.primal(x1) << std::endl;
std::cout << "x2 = " << lp.primal(x2) << std::endl;
```

The LP solvers are powerful general tools for solving various complex optimization problems. Let us consider the well-known maximum flow problem for example. It is to find a flow of maximum value between a source and a target node in a network with capacity constraints. Let $G = (V, A)$ denote a digraph, let $c : A \to \mathbb{R}^+$ denote a capacity function and let $s, t \in V$ denote the source and target nodes, respectively. A maximum flow is an $f : A \to \mathbb{R}$ solution of the following optimization problem.

$$\begin{aligned} \max \quad & \sum_{a \in \mathrm{A_{out}}(s)} f_a - \sum_{a \in \mathrm{A_{in}}(s)} f_a & \\ \text{s. t.} \quad & \sum_{a \in \mathrm{A_{out}}(u)} f_a = \sum_{a \in \mathrm{A_{in}}(u)} f_a & \forall u \in V \setminus \{s, t\} \\ & 0 \leq f_a \leq c_a & \forall a \in A \end{aligned}$$

The task can be solved in LEMON with the following code:

```
Lp lp;
GR::ArcMap<Lp::Col> f(g);
lp.addColSet(f);

// Objective function
Lp::Expr obj;
for (GR::OutArcIt a(g, src); a != INVALID; ++a) obj += f[a];
for (GR::InArcIt a(g, src); a != INVALID; ++a)  obj -= f[a];
```

```
lp.max();
lp.obj(obj);

// Flow conservation constraints
for (GR::NodeIt v(g); v != INVALID; ++v) {
  if (v == s || v == t) continue;
  Lp::Expr expr;
  for (GR::OutArcIt a(g, v); a != INVALID; ++a) expr += f[a];
  for (GR::InArcIt a(g, v); a != INVALID; ++a)  expr -= f[a];
  lp.addRow(expr == 0);
}

// Capacity constraints
for (GR::ArcIt a(g); a != INVALID; ++a) {
  lp.colLowerBound(f[a], 0);
  lp.colUpperBound(f[a], c[a]);
}

// Solve LP
lp.solve();
```

Note that the expressions are built using simple loops that traverse the outgoing and incoming arcs of nodes. Various other graph optimization problems can also be expressed as linear programs and the interface provided in LEMON facilitates solving them easily (though usually not so efficiently as by a direct combinatorial method, if one exists).

Currently, the following linear and mixed integer programming packages are supported by LEMON: GLPK [32], Clp [10], Cbc [8], ILOG CPLEX [14], and SoPlex [62]. Additional wrapper classes for new solvers can also be implemented quite easily.

5.2 Implementation Details

This section presents selected implementation details of LEMON along with specific code examples that demonstrate the applied techniques.

5.2.1 Adjacency Lists in Vectors

The general graph types of LEMON store the adjacency lists internally in `std::vector`s and use the vector indices as identifiers of the nodes and arcs. `Node` and `Arc` objects store these indices, thus for each of them, the corresponding vector element can be looked up in constant time.

For example, the following code fragment comes from the source of `SmartDigraph`.

```
struct NodeT {
  int first_in;        // index of the first incoming arc
  int first_out;       // index of the first outgoing arc
};
struct ArcT {
  int target, source;  // indices of the endnodes
  int next_in;         // index of the next incoming arc
  int next_out;        // index of the next outgoing arc
};
std::vector<NodeT> nodes;
std::vector<ArcT> arcs;
```

The iteration on the outgoing arcs of a given node begins with the lookup of the corresponding `NodeT` item, whose `first_out` member stores the index of the first arc. After that, each step reads the `next_out` value from the current `ArcT` object to obtain the index of the next arc, or `-1` if the current arc is the last one. The incoming arcs are handled in the same way using the members `first_in` and `next_in`. It means that the incident arcs are recorded using simply-linked lists that are actually stored in a vector.

`ListDigraph` is implemented similarly, but it maintains the nodes and arcs using doubly-linked lists to support the efficient deletion of them, as well as the addition of new elements.

A major advantage of these data structures is that each node and arc is associated with a unique integer identifier, which is a crucial requirement for implementing efficient external maps (see Section 5.1.3). On the other hand, this design restricts the customization possibilities of the graph data structures, because the nodes and arcs must be stored in random access containers.

BGL applies another approach: its main graph type, the `adjacency_list` class is highly customizable. Container types can be specified by template arguments separately for the node list and the incident edge lists of the nodes. Using advanced data structures for these purposes can be beneficial in certain cases. For example, storing the incident edges in an `std::set` allows logarithmic time lookup of an outgoing edge with a given target node. However, the lack of naturally assigned unique integer identifiers makes it harder to use external maps, which are frequently required in graph algorithms for temporary storage.

5.2.2 Extending Graph Interfaces Using Mixins

A fundamental problem of designing a general graph concept is that an easy-to-implement concept should require the least number of overlapping functionality, but this approach strongly limits the versatility of the interface. This contradiction is overcome by developing two-level graph concepts.

In LEMON, the *user-level* graph concepts define a wide range of member functions and nested classes, and so they support convenient and flexible use. On the other hand, the *low-level* graph concepts define only the very basic functionality, for example, simplified function-based iteration. These simple interfaces are extended to the user-level concepts using the template *Mixin* strategy [61]. Specifically, if a class `DigraphBase` implements the low-level interface, then `DigraphExtender<DigraphBase>` will satisfy the requirements of the user-level `Digraph` concept.

```
class DigraphBase {
public:
  // Node and Arc classes
  class Node { ... };
  class Arc { ... };

  // Basic iteration
  void first(Node& node) const;
  void next(Node& node) const;
  ...
};
```

The extender adds nested iterator and map classes to the graph type, as well as the required member variables for alteration observing (see Section 5.2.3). If the underlying graph class also defines functions for node and arc addition or deletion, then they are overridden to handle the alteration observing, as well.

```
template <typename DigraphBase>
class DigraphExtender : public DigraphBase {
public:
  // Iterator class
  class NodeIt : public Node {
  public:
    NodeIt(const DigraphExtender& g) : _graph(g) {
      _graph.first(*this);
    }
    NodeIt& operator++() {
      _graph.next(*this);
```

```
      return *this;
    }
    ...
  private:
    const DigraphExtender& _graph;
  };
  ...
};
```

5.2.3 Signaling Graph Alterations

LEMON graph maps are external, auto-updated data structures. They are implemented using arrays or `std::vector`s to ensure efficient data access, which is the most important design goal of maps. However, these data structures are extended when new nodes or arcs are added to the associated graph.

The graph and map types implement the *Observer* design pattern [27], they signal the changes of the node and arc sets. The observed events are limited to adding and removing one or several items, building the graph from scratch, and removing all items from it. The observers are inherited from the corresponding `AlterationNotifier<Graph, Item>::ObserverBase` class, and they have to override the event handler functions.

Graphs contain instances of `AlterationNotifier<C, I>` for each item type.

```
class Graph {
  ...
protected:
  AlterationNotifier<Graph, Node> _node_notifier;
  AlterationNotifier<Graph, Arc> _arc_notifier;
  AlterationNotifier<Graph, Edge> _edge_notifier;
};
```

The graph maps are designed to be exception safe. In fact, they guarantee strong exception safety [63]. If a node or arc is inserted into a graph, but an attached map cannot be extended, then each map extended earlier is rolled back to its original state.

5.2.4 Tags and Specializations

The functionality and efficiency of generic libraries can be further improved by template specializations. In LEMON, *tags* are defined for several purposes. For instance, the graphs are marked with `UndirectedTag`.

```
class ListDigraph {
  typedef False UndirectedTag;
  ...
};
class ListGraph {
  typedef True UndirectedTag;
  ...
};
```

Let us consider the checking of the Eulerian property. A directed graph is Eulerian if it is connected and the number of incoming and outgoing arcs are the same for each node. On the other hand, an undirected graph is Eulerian if it is connected and the number of incident edges is even for each node. Therefore, the `eulerian()` function is specialized for undirected graphs using `UndirectedTag` as follows.

```
template <typename GR>
typename enable_if<typename GR::UndirectedTag, bool>::type
eulerian(const GR &g) {
  for (typename GR::NodeIt v(g); v != INVALID; ++v) {
    if (countIncEdges(g, v) % 2 == 1) return false;
  }
  return connected(g);
}
```

LEMON uses bool-valued tags and `enable_if` borrowed from the Boost libraries [4, 36, 73] to implement the specializations. This technique allows more options in combination of rules than the simple tag-based dispatching.

Another example for specialization can be found in the implementation of graph maps. Because the data vectors of `ListDigraph` and the corresponding maps could contain gaps, some items shall not be constructed. To avoid the unnecessary data initializations and potential side effects, the values of the maps are constructed with placement `new` when items are inserted into the graph. However, maps of POD value types are implemented with `std::vector`s because their constructors are for free and do not have side effects. The values are reset when the items are removed from the graph.

5.2.5 Concept Checking

Concept checking is an important part of generic programming, because it is worthwhile to check the conformance of the template parameters of each class and function template. In addition, the error messages become clearer and understandable, and they point to the location of the error-prone instantiation and not to the internal implementation.

In LEMON, the concept checking class and the archetype for a particular concept are encapsulated into the same class, and it deals with the interface without any actual functionality. It also contains an additional `Constraint` struct, which is responsible for the concept checking. Its `constraints()` function tests the interface, it have to be instantiable with all realizations of the concept.

```
template <typename PR, typename IM>
class Heap {
public:
  void push(const IM::Key &i, const PR &p) {}
  ...
  template <typename _Heap>
  struct Constraints {
    void constraints() {
      heap.push(typename IM::Key(), PR());
      ...
    }
    _Heap& heap;
  };
};
```

The LEMON concept checking is based on the Boost Concept Check Library, but it is simplified and uses less precompiler macros. The introduction of native language support for concepts were planned for several years but finally, it was decoupled from the latest c++11 standard [64].

5.3 Performance

This section compares the running time performance of LEMON to BGL and LEDA. The experiments were conducted using LEMON 1.2, Boost 1.48.0, and LEDA 5.0.

Three fundamental problems were considered in the tests:

- finding shortest paths from a designated source node in a graph with non-negative arc lengths;
- finding maximum cardinality matching in general graphs;
- planarity checking of simple graphs.

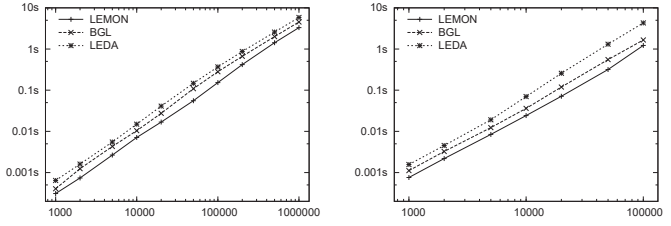
The benchmark tests were performed on a machine with AMD Opteron Dual Core 2.2 GHz CPU and 16 GB memory (1 MB cache), running

openSUSE 11.4 operating system. The codes were compiled with GCC version 4.5.3 using -O3 optimization flag.

The test instances for the Dijkstra algorithm [13] were created with NETGEN [38], a popular generator for various network problems. Two different benchmark suites are considered. The first one contains sparse graphs, for which m is about $n \log_2 n$, where n and m denote the number of nodes and arcs, respectively. In the second set, there are networks for which m is roughly $n\sqrt{n}$, so they are relatively dense. The arc capacities and costs were generated evenly from the range [1..10000].

The charts in the following figures show the measured running times in seconds as a function of the number of nodes in the graph. Logarithmic scale is used for both axes to ensure suitable diagrams. In these experiments, the most efficient general graph data structure was used for each library. Namely, `SmartDigraph` was used for LEMON, `adjacency_list<vecS, vecS, directedS, ...>` was used for BGL and `graph` was used for LEDA.

Figure 5.3 shows the benchmark results for finding shortest paths. All the three libraries implement Dijkstra's algorithm for this problem with several priority queue representations. To obtain comparable results, the standard binary heap data structure was selected for each library. BGL was more efficient than LEDA, especially on large dense graphs, for which it turned out to be more than two times faster. However, LEMON performed significantly better than both of them on all problem instances. Since Dijkstra's algorithm is rather simple, these differences were obviously induced by the efficiency of the applied graph, map, and heap data structures.



Sparse graphs ($m \approx n \log_2 n$) Dense graphs ($m \approx n\sqrt{n}$)

Figure 5.3: Benchmark results for the Dijkstra algorithm

Since LEMON and BGL are generic and open source libraries, we could implement graph adaptor classes that make it possible to run LEMON algorithms on BGL graph data structures and BGL algorithms on LEMON graph

data structures. Table 5.1 contains benchmark results of such comparisons. The performance of the Dijkstra algorithm is measured on the largest problem instances for all combinations of the LEMON and BGL implementations (`SmartDigraph` and `adjacency_list<vecS, vecS, directedS, ...>` were used as before). The binary heap data structures were considered as parts of the algorithm implementations. However, the property maps are strongly related to the graph data structures, thus they were exchanged together with the graphs. Note that the differences in the design decisions of the libraries could have a huge effect on the performance of fundamental data structures (see Sections 5.1.1, 5.1.3, 5.2.1).

Graph type	Algorithm	Sparse graph	Dense graph
LEMON	LEMON	3.30s	1.22s
LEMON	BGL	4.23s	0.98s
BGL	LEMON	3.29s	1.41s
BGL	BGL	4.53s	1.66s

Table 5.1: Benchmark results for the largest instances of the shortest path problem combining LEMON and BGL implementations

These results show that LEMON's `SmartDigraph` implementation is significantly faster than the `adjacency_list` data structure of BGL. Moreover, the Dijkstra algorithm of LEMON also proved to be more efficient, probably because of the better implementation of the heap data structure. The BGL graph type with the BGL algorithm implementation was clearly the slowest combination.

Apart from the general graph types, all the three libraries provide more efficient static graph implementations, which were also tested. Table 5.2 compares the performance of Dijkstra's algorithm using the general and static graph types of the libraries. The main conclusion of these results is that LEMON's `SmartDigraph` implementation was almost as efficient as `StaticDigraph`, while the general graph types of the other two libraries turned out to be much slower than the static representations, especially when relatively dense graphs are considered. We can also note that the differences between the libraries were smaller using the optimized graph representations. For dense graphs, the running times were practically the same.

This comparison fairly demonstrates the importance of efficient graph data structures and their effect on the overall performance of algorithms. Although the static graph types are clearly more efficient, the performance of general graph types is also important because they are used more frequently.

The performance results for the maximum cardinality matching problem instances are presented in Figure 5.4. Each library provides an implemen-

Implementation	Sparse graph	Dense graph
LEMON with `SmartDigraph`	3.30s	1.22s
LEMON with `StaticDigraph`	3.26s	0.85s
BGL with `adjacency_list`	4.53s	1.66s
BGL with `compressed_sparse_row_graph`	4.56s	0.97s
LEDA with `graph`	5.85s	4.30s
LEDA with `static_graph`	3.95s	0.80s

Table 5.2: Benchmark results for the largest instances of the shortest path problem using general and static graph types

tation of Edmond's maximum cardinality matching algorithm [22]. We used two sets of Erdős-Rényi random graphs. The first set contains sparse graphs, for which m is equal to n. In the second one, there are networks for which m is roughly $n \log_2 n$, so they are relatively dense for this problem.

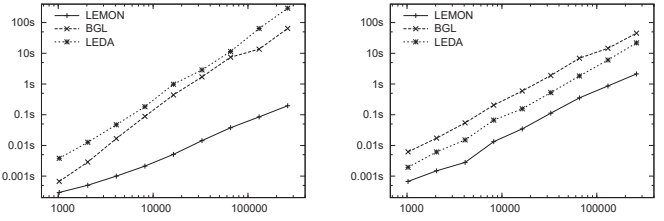
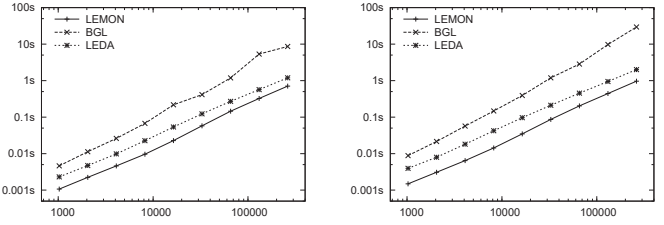


Figure 5.4: Benchmark results for maximum matching algorithms

All of the three implementations use Edmond's matching algorithm with greedy initialization. For dense graphs, BGL is more efficient than LEDA, but for sparse graphs, we get opposite result. However, LEMON is significantly more efficient than both of the other two libraries. The difference is bigger for sparse graphs, which represent harder instances for this problem. In dense graphs, the greedy initialization can find a relatively large matching, which decreases the number of iterations in Edmond's algorithm. In LEMON, this step is implemented more efficiently, which causes bigger difference in the performance for the sparse benchmark set.

The third optimization problem we considered is the planarity checking of undirected graphs. Both LEMON and BGL implement the Boyer-Myrvold algorithm [6], while LEDA uses Hopcroft-Tarjan planar embedding method [34].

Two sets of graphs are used to benchmark planarity checking, as well. The first set contains relatively dense graphs. Random point sets are generated uniformly in a disc, and the Delaunay-triangulations of the point sets are calculated and stored as a graph. Each of these graphs contains $m = 3n-3-n_{\text{ext}}$ edges, where n_{ext} is the number of vertices of the point set's convex hull. The second benchmark set is also generated with Delaunay-triangulation, but then the half of the edges are removed from the graph with maintaining the bi-connectivity.



Sparse graphs ($m \approx 1.5n$) Dense graphs ($m \approx 3n$)

Figure 5.5: Benchmark results for planarity checking algorithms

For the planarity checking problem, LEMON provides the best performance for both sparse and dense graphs. LEDA outperformed the BGL implementation, despite the fact that BGL uses the same algorithm as LEMON.

These results and various other experiments imply the conclusion that the fundamental algorithms and data structures of LEMON turned out to be measurably faster than the corresponding implementations of the other two libraries. This achievement is clearly one of the most important benefits of LEMON, and it could be a major reason for using this library.

5.4 Conclusions

LEMON is a highly efficient, open source C++ graph template library having a clear design and convenient interface.

- The library provides a wide range of data structures, algorithms and other practical components, which can be combined easily for solving problems of various types related to graphs and networks.
- According to extensive benchmark tests, essential algorithms and data structures of LEMON typically turned out to be more efficient than

the corresponding tools of widely used similar libraries, namely BGL and LEDA.

- The author has made many contributions to the library including the development of several algorithms and data structures. Some significant design decisions can also be mentioned among his activities.
- The wide toolkit of the library makes it also available to use in several optimization algorithms related to geoinformatics and remote sensing.
- The LEMON graph data structures can naturally represent objects of real world geographic systems. Furthermore, due to the grid graph implementation, LEMON is easy to use in image processing algorithms.

6 Summary

The dissertation studies three real-world problems and related optimization algorithms. In addition, the LEMON software library is also presented, which has played an important role during solving the former three problems.

In Section 2, the satellite image classification is studied, and the performance of four graph-based segmentation algorithms are compared in the classification problem. It was shown that the segment-based algorithms can outperform the pixel-wise algorithms, and the top-down segmentation algorithms usually provide better performance than the bottom-up segmentation methods. In practice, the minimum normalized cut segmentation with Bhattacharyya-distance based classification provides a competitive option.

In Section 3, the raster-vector conversion of topographic maps is discussed. A framework was developed for the vectorization process, which contains several image processing algorithms and a customized workflow was created for Hungarian topographic map types.

In Section 4, the problem of optimizing working schedules is discussed. An integer programming formulation was given for the problem and a column generation-based linear programming algorithm was developed for solving it.

In Section 5, the LEMON library is introduced, which has been used as a basic toolkit for implementing the algorithms for the former problems.

When a complex real-world problem is to be solved, mathematics, computer science and software technology should be used in synthesis. First, the mathematical model should be established, then an adequate optimization method should be elaborated, and finally efficient algorithms should be implemented using appropriate software development environment. This approach has been successfully used to solve the above problems.

References

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, 2009. 12

[2] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007. 12

[3] Jean-Marie Beaulieu and Morris Goldberg. Hierarchy in picture segmentation: A stepwise optimization approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(2):150–163, 1989. 17

[4] Boost C++ Libraries. `http://www.boost.org/`, 2012. 58, 74

[5] Ralf Borndörfer, Uwe Schelten, Thomas Schlechte, and Steffen Weider. A column generation approach to airline crew scheduling. Technical report, Konrad Zuse Institute, Berlin, 2005. 53

[6] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified $o(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004. 78

[7] Ruini Cao and Chew L. Tan. Text/graphics separation in maps. In *GREC '01: Selected Papers from the Fourth International Workshop on Graphics Recognition Algorithms and Applications*. Springer-Verlag, 2002. 30

[8] Cbc – Coin-Or Branch and Cut. `http://projects.coin-or.org/Cbc/`, 2012. 70

[9] Fan Chung. *Spectral Graph Theory*. Number 92 in CBMS Regional Conf. Series in Mathematics. American Mathematical Society, 1997. 22

[10] Clp – Coin-Or Linear Programming. `http://projects.coin-or.org/Clp/`, 2012. 70

[11] COIN-OR - COmputational INfrastructure for Operations Research. `http://www.coin-or.org/`. 56

[12] William Cook and Andre Rohe. Computing minimum-weight perfect matchings. *INFORMS J. on Computing*, 11(2):138–148, 1999. 20

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001. 50, 76

[14] ILOG CPLEX. `http://www.ilog.com/`, 2012. 68, 70

[15] Balázs Dezső, István Fekete, Dávid Gera, Roberto Giachetta, and István László. Object-based image analysis in remote sensing applications using various segmentation techniques. *Annales Univ. Sci. Budapest, Sect. Comp*, 36, 2012. Accepted for publication. 13

[16] Balázs Dezső, Roberto Giachetta, László István, and Fekete István. Experimental study on graph-based image segmentation methods in the classification of satellite images. *EARSeL eProceedings*, 11(1):12 – 24, 2012. 23

[17] Balázs Dezső, Alpár Jüttner, and Péter Kovács. Column generation method for an agent scheduling problem. *Electronic Notes in Discrete Mathematics*, 36:829–836, 2010. ISCO 2010 - International Symposium on Combinatorial Optimization. 45

[18] Balázs Dezső, Alpár Jüttner, and Péter Kovács. LEMON – an Open Source C++ Graph Template Library. *Electronic Notes in Theoretical Computer Science*, 264(5):23 – 45, 2011. Proceedings of the Second Workshop on Generative Technologies (WGT) 2010. 56

[19] Balázs Dezső, Zsigmond Máriás, and István Elek. Image processing methods in raster-vector conversion of topographic maps. In *International Conference on Artificial Intelligence and Pattern Recognition*, July 2009. 29

[20] Stuart E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, pages 395–412, 1969. 51

[21] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification and scene analysis*. Wiley, 1996. 12

[22] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965. 78

[23] EGRES - egerváry research group on combinatorial optimization. `http://www.cs.elte.hu/egres/`. 56

[24] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *Int. J. Comput. Vision*, 59(2):167–181, 2004. 17

[25] Steven Fortune. A sweepline algorithm for voronoi diagrams. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 313–322, New York, NY, USA, 1986. ACM. 40

[26] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *SODA '90: Proc. of the first annual ACM-SIAM symp. on Discrete algorithms*, pages 434–443, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics. 20

[27] Erich Gamma, Richard Helm, Ralph Johnson, and Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994. 73

[28] Scott Kirkpatrick, C.D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. 35

[29] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984. 13, 35

[30] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002. 58, 64

[31] Roberto Giachetta. Gráf alapú módszerek műholdfelvételek tematikus osztályozásában. 25$^{\text{th}}$ National Scientific Student Conference, Informatics Section, 2009. 3$^{\text{rd}}$ prize. 9

[32] GLPK – GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`, 2012. 53, 70

[33] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. 51, 52

[34] John Hopcroft and Robert E. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974. 78

[35] Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. *Column Generation*, pages 33–65, 2005. 49

[36] Jaakko Järvi, Jeremiah Willcock, H. Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, pages 25–32, June 2003. 74

[37] Alireza Khotanzad and Edmund Zink. Contour line and geographic feature extraction from USGS color topographical paper maps. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(1):18–31, 2003. 30

[38] Darwin Klingman, Albert Napier, and Joel Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20:814–821, 1974. 76

[39] Yao-Yi Chiang, Craig A. Knoblock, and Chen Ching-Chien. Automatic extraction of road intersections from raster maps. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 267–276, New York, NY, USA, 2005. ACM. 30

[40] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, NY, USA, 1993. 62

[41] Vladimir Kolmogorov. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1(1), 2009. 20

[42] NASA portal – the LANDSAT program. `http://landsat.gsfc.nasa.gov/`, 2012. 10

[43] István László, Balázs Dezső, István Fekete, and Tamás Pröhle. A fully segment-based method for the classification of satellite images. *Annales Univ. Sci. Budapest, Sect. Comp*, 31, 2009. 12, 23

[44] István László, Katalin Ócsai, Dávid Gera, Roberto Giachetta, and István Fekete. Object-based image analysis of pasture with trees and red mud spill. In *Proceedings of the 31th EARSeL Symposium*, 2011. 13

[45] István László, Tamás Pröhle, István Fekete, and Gábor Csornai. A method for classifying satellite images using segments. *Annales Univ. Sci. Budapest, Sect. Comp*, 23, 2004. 12

[46] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *SFCS '88: Proc. of the 29th Annual Symp.*

on Foundations of Computer Science, pages 422–431, Washington, DC, USA, 1988. IEEE Computer Society. 21

[47] LEMON – Library for Efficient Modeling and Optimization in Networks. `http://lemon.cs.elte.hu/`, 2012. 21, 56

[48] Serguei Levachkine. Raster to vector conversion of color cartographic maps. In *LNCS, Graphics Recognition*. Springer-Verlag, 2004. 29

[49] Marco E. Lübbecke and Jacques Desrosiers. Selected topics in column generation. *Oper. Res.*, 53(6):1007–1023, 2005. 47

[50] István Elek, Zsigmond Máriás, and Balázs Dezső. IRIS, Intelligent Raster Interpretation System (in Hungarian). Technical report, Faculty of Informatics, Eötvös Loránd University, Budapest, 2007. 29

[51] Kurt Mehlhorn and Stefan Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, New York, NY, USA, 1999. 58

[52] Kurt Mehlhorn and Guido Schäfer. Implementation of $O(nm \log(n))$ weighted matchings in general graphs: the power of data structures. *J. Exp. Algorithmics*, 7:4, 2002. 20

[53] Carolyn Habit Norton, Serge A. Plotkin, and Éva Tardos. Using separation algorithms in fixed dimension. In *SODA '90: Proc. of the first annual ACM-SIAM symp. on Discrete algorithms*, pages 377–387, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics. 20

[54] National remote sensing centre – indian space research organization. `http://www.nrsc.gov.in`, 2012. 10

[55] James K. Park and Cynthia A. Phillips. Finding minimum-quotient cuts in planar graphs. In *STOC '93: Proc. of the 25th annual ACM symp. on Theory of computing*, pages 766–775, New York, NY, USA, 1993. ACM. 21

[56] John A. Richards and Xiuping Jia. *Remote sensing digital image analysis*. Springer Berlin etc., 1993. 9

[57] Stuart J. Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Prentice hall, 2010. 51

[58] Spinello Salvatore and Pascal Guitton. Contour lines recognition. *Journal of WSCG*, 12, 2004. 30

[59] SGB – Stanford GraphBase. `ftp://ftp.cs.stanford.edu/pub/sgb/`, 2012. 62

[60] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000. 21

[61] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 163–177, London, UK, 2001. Springer-Verlag. 72

[62] SoPlex – The Sequential Object-Oriented Simplex. `http://soplex.zib.de/`, 2012. 70

[63] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition, February 2000. 73

[64] Bjarne Stroustrup. No 'concepts' in c++0x. `http://accu.org/index.php/journals/1576`, 2009. 75

[65] Rudolf Szendrei, István Elek, and Mátyás Márton. A knowledge-based approach to raster-vector conversion of large scale topographic maps. *Acta Cybernetica*, 20:145–165, 2011. 29

[66] Robert E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983. 17, 18

[67] James C. Tilton. Image segmentation by iterative parallel region growing and splitting. In *Proc. of the International Geoscience and Remote Sensing Symp. (IGARSS89)*, pages 2235–2238, 1989. 17

[68] Godfried Toussaint. Solving geometric problems with the rotating calipers. In *In Proc. IEEE MELECON '83*, 1983. 43

[69] Aurelio Velázquez and Serguei Levachkine. Text/graphics separation and recognition in raster-scanned color cartographic maps. In *LNCS, Graphics Recognition*. Springer-Verlag, 2004. 30

[70] Sheng-Guo Wang and Yuanlu Bao. New intelligent method to generate vector maps for GPS navigation. In *Proceedings of the 15th IFAC World Congress, 2002*, 2002. 30

[71] Song Wang and Jeffrey Mark Siskind. Image segmentation with minimum mean cut. *8^{th} IEEE International Conf. on Computer Vision*, 1:517, 2001. 18, 19

[72] Song Wang and Jeffrey Mark Siskind. Image segmentation with ratio cut. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 25(6):675–690, 2003. 18, 19

[73] István Zólyomi and Zoltán Porkoláb. Towards a general template introspection library. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering, Third International Conference, GPCE 2004*, volume 3286 of *Lecture Notes in Computer Science*, pages 266–282, Vancouver, Canada, October 2004. Springer. 74

[74] William K. Pratt. *Digital Image Processing.* John Wiley & Sons, New York, 1991. 31

[75] LEDA – Library of Efficient Data Types and Algorithms. `http://www.algorithmic-solutions.com/`, 2010. 58